USBORNE INTRODUCTION TO

MACHINE CODE FOR BEGINNERS

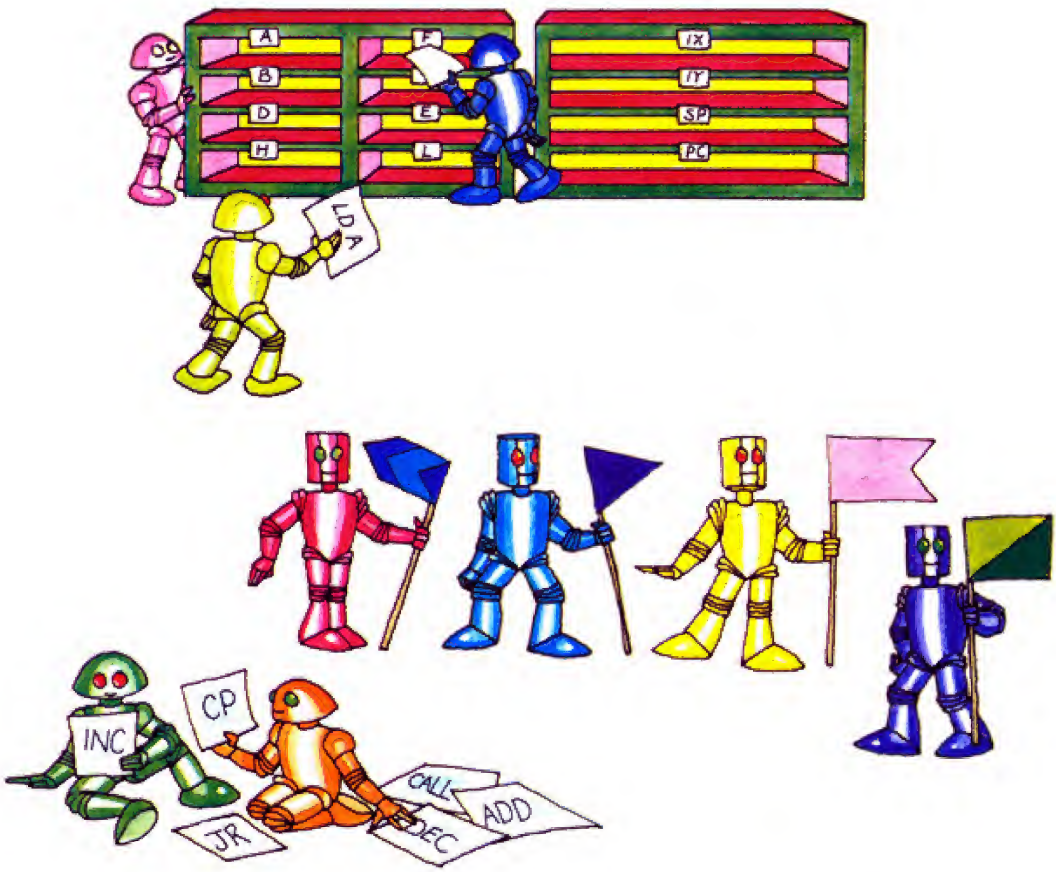
FOR THE Z80 AND 6502

Usborne Computer Books

USBORNE INTRODUCTION TO

MACHINE CODE FOR BEGINNERS

Lisa Watts and Mike Wharton



Illustrated by Naomi Reed and Graham Round
Designed by Graham Round and Lynne Norman
6502 consultants: A. P. Stephenson and Chris Oxlade

Contents

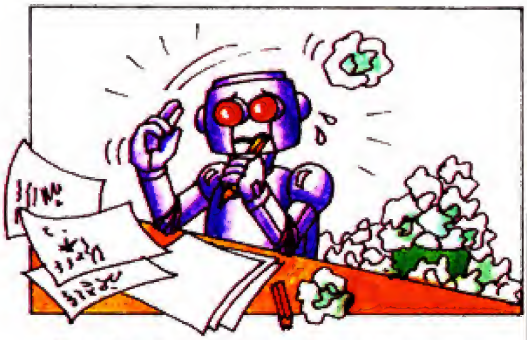
4 What is machine code?
6 Getting to know your computer
8 The computer's memory
11 Hex numbers
12 Peeking and poking
14 Inside the CPU
16 Giving the CPU instructions
18 Translating a program into hex
20 Finding free RAM
23 Loading and running a program
27 Adding bytes from memory
28 Working with big numbers
29 The carry flag
30 Big number programs
32 Displaying a message on the screen
35 Jumping and branching
38 Screen flash program
40 Going further
41 Decimal/hex conversion charts
42 Z80 mnemonics and hex codes
45 6502 mnemonics and hex codes
46 Machine code words
48 Index

First published 1983 by Usborne Publishing Ltd, 20 Garrick Street, London WC2E 9BJ, England.

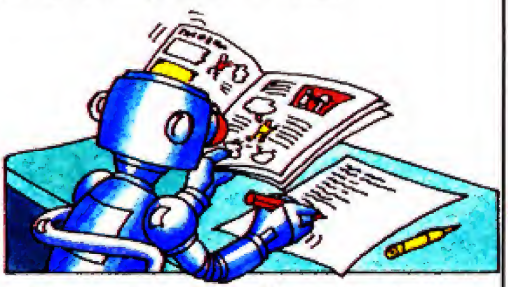
Printed in Spain by Printer Industria Gráfica, S. A. - Depósito Legal B. 33.755/1983

About this book

This book is a simple, step-by-step guide to learning to program in machine code. Machine code is the code in which the computer does all its work and programs written in machine code run much faster and take up less memory space than programs in BASIC. A machine code program, though, is much more difficult to write and less easy to understand than a program in BASIC.

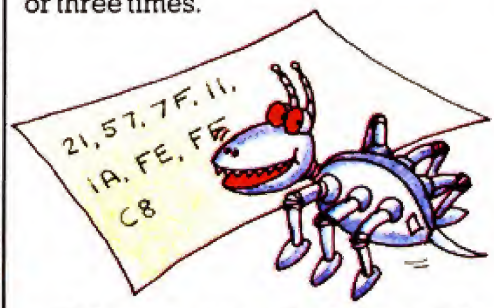


This book takes you in very easy stages through the basic principles of machine code. It shows you how to write simple machine code programs, for example, to add two numbers or flash a message on the screen, and how to load and run a machine code program on your computer.

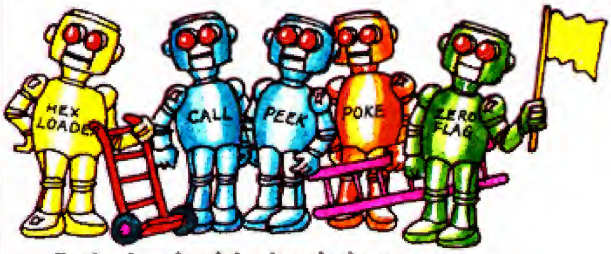


The book is specially written for computers with a Z80 or 6502 microprocessor.* The microprocessor is the chip which contains the computer's central processing unit and computers with different microprocessors understand different machine code. All computers with the same type of microprocessor, though, use the same machine code.

Machine code is difficult and very laborious, with lots of rules to obey and small details to remember. Don't worry if you find it very hard at first. It seems confusing as you cannot read and understand a program in machine code – it's just a string of letters and numbers. Bugs are very difficult to spot, too, and have disastrous results if you miss them. When you are working in machine code you have to be very careful and methodical and check everything two or three times.



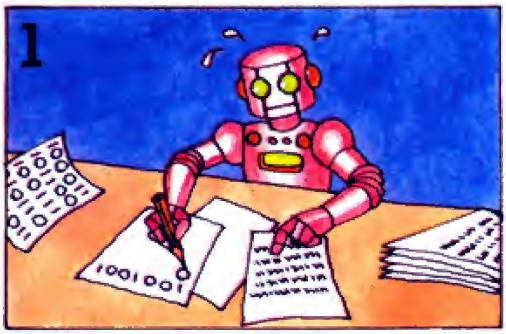
Unless you are really dedicated there is no point in writing long programs in machine code – some things can be done just as well in BASIC. For certain tasks, though, such as speeding up the action in games programs or creating fantastic screen effects, you need to use machine code. This book shows you how to make your programs more exciting by using short machine code subroutines in BASIC programs.



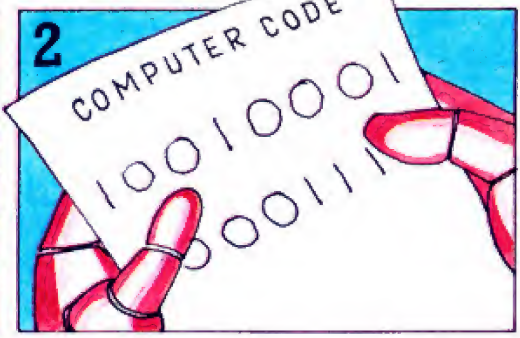
At the back of the book there are some conversion charts to help you when you are writing machine code, and a list of machine code words to explain all the jargon. There are also lots of puzzles and ideas for short programs to write, with answers on page 44.

*The Spectrum and ZX81 (Timex 2000 and 1000) use the Z80 microprocessor and the VIC 20, the BBC, the Atari computers and the Oric use the 6502. The Commodore 64 uses the 6510 and understands 6502 machine code.

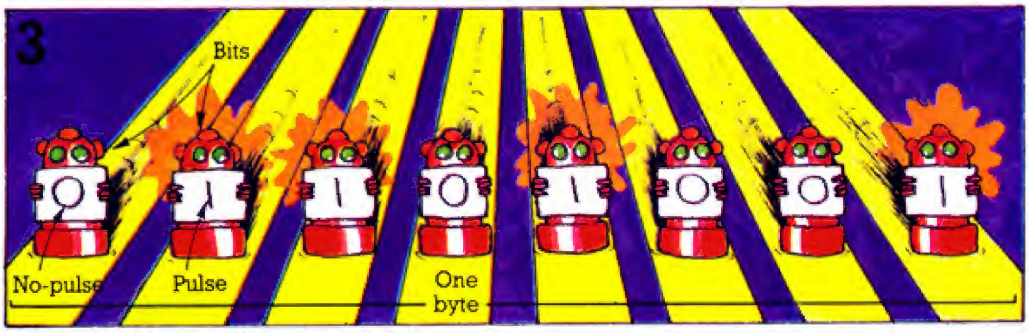
What is machine code?



Machine code is the code in which the computer does all its work. When you give a computer a program in BASIC, all the instructions and data are translated into machine code inside the computer.

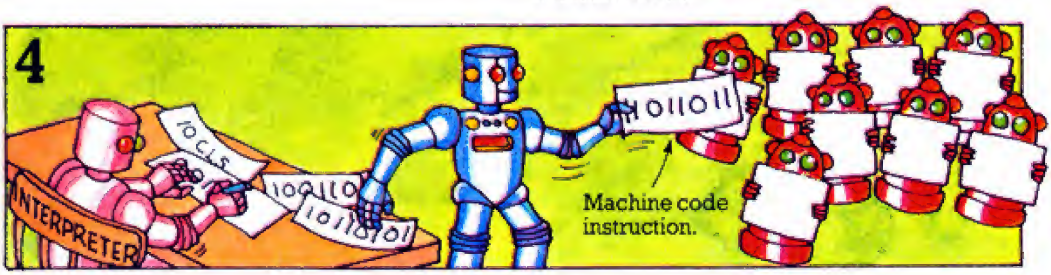


In machine code, each instruction and piece of information is represented by a binary number. Binary is a number system which uses only two digits, 1 and 0. You can write any number in binary using 1s and 0s.*



Inside the computer, the binary numbers are represented by pulses of electricity, with a pulse for a 1 and no pulse for a 0. The pulses and no-pulses are called "bits", short for binary digits.

The bits flow through the computer in groups of eight and each group is called a "byte". Each byte of pulses and no-pulses represents the binary number for one instruction or piece of information in machine code.



Each task the computer can carry out, such as adding two numbers or clearing the screen, involves a sequence of several instructions in machine code. When you give the computer a BASIC command, a special program called the "interpreter" translates your command into the machine code instructions the computer understands.

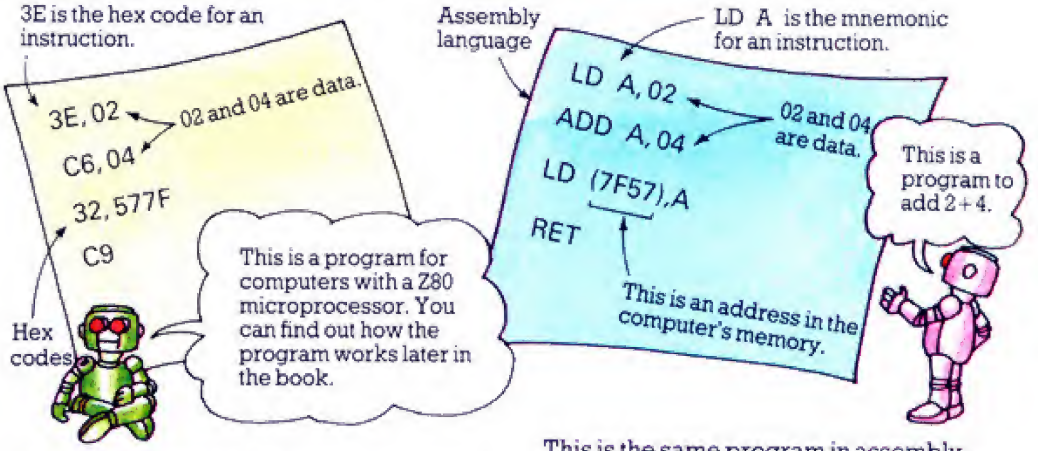
The term machine code is also used to refer to programs written in a form which is much closer to the computer's code than BASIC is. In a machine code program you have to give the computer all the separate instructions it needs to carry out a task such as clearing the screen.

*You can find out more about binary on page 28.

Programming in machine code

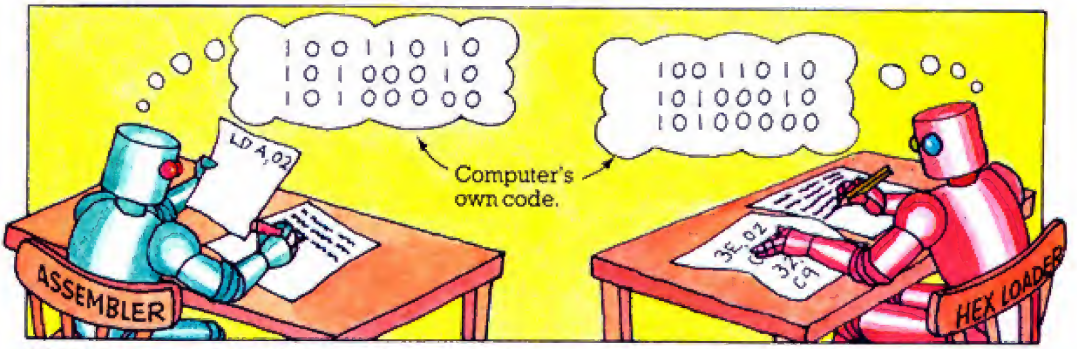
There are several different ways of writing machine code programs. You could write all the instructions in binary numbers, but this would be very tedious. Instead, you can use another number system called hex, short for hexadecimal. Once you get used to it, hex is much easier to work with than binary.

Machine code programs can also be written in a code called "assembly language". In assembly language each instruction to the computer is represented by a "mnemonic" (pronounced nemonic) – a short word which sounds like the instruction it represents.



This is part of a machine code program in hex. The hex number system has sixteen digits and uses the symbols 0-9 and A-F to represent the numbers 0 to 15. (You can find out more about hex later in the book.) The hex number at the beginning of each line of the program is an instruction (e.g. 3E). It is the hex equivalent of the binary code for that instruction.

This is the same program in assembly language. Each line contains the mnemonic for one instruction and is the equivalent of the hex number in the same line on the left. For example, the mnemonic LD A (pronounced "load A") means the same as the hex number 3E. In both these programs, each line contains an instruction which is the equivalent of a single instruction in the computer's own code.



To give a computer a program in assembly language you need a special program called an "assembler" which translates the mnemonics into the computer's code. Some computers have a built-in assembler; with others, you can buy an assembler on cassette and load it into the computer's memory. Alternatively you can write a machine code program using the mnemonics of assembly language (they are easier to remember than numbers), then translate them into hex before you give them to the computer. Some computers will accept hex numbers; with others you have to give them a short program, called a "hex loader", which translates them for the computer. There is a hex loader program on page 24 which you can use to load the machine code programs in this book.

Getting to know your computer

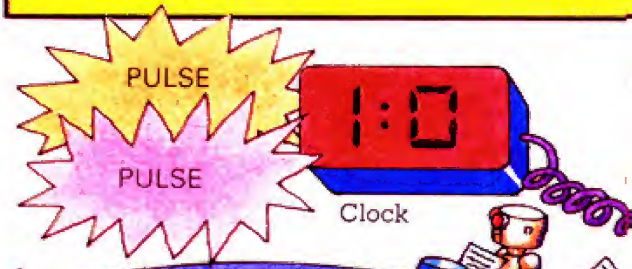
When you program a computer in machine code you have to tell it exactly what to do at each stage: where to find and store data, how to print on the screen and so on. (When you are working in BASIC, special programs inside the computer take care of all this for you.) In order to give the computer the correct machine code instructions, you need a good idea of what is going on inside your computer. The pictures on these two pages show the parts inside a home computer, and what they are for. You can find out more about them on the next few pages.

Inside a computer

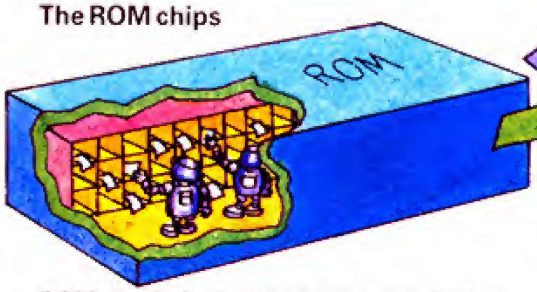
Inside the keyboard of a microcomputer there is a printed circuit board. This has metal tracks printed on it, along which electric currents can flow. Attached to the printed circuit board there are a number of chips.

What the chips do

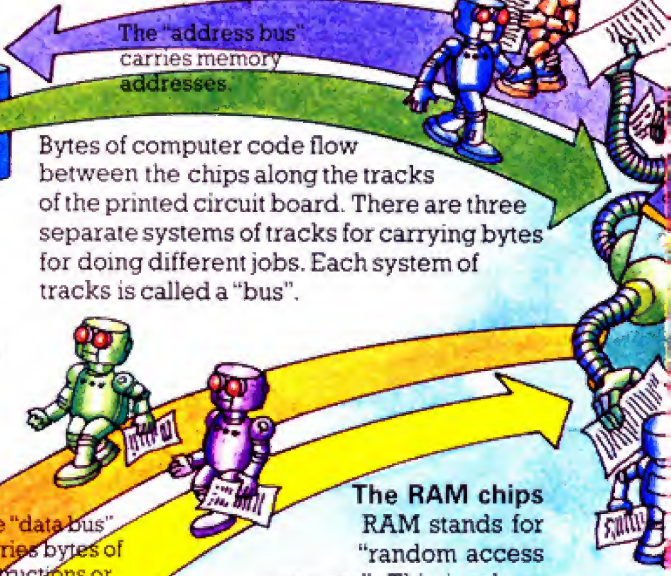
This picture shows the work carried out by the different chips inside the computer. Messages flow between the chips in the form of bytes, i.e. groups of eight pulse and no-pulse signals representing data and instructions.



The ROM chips

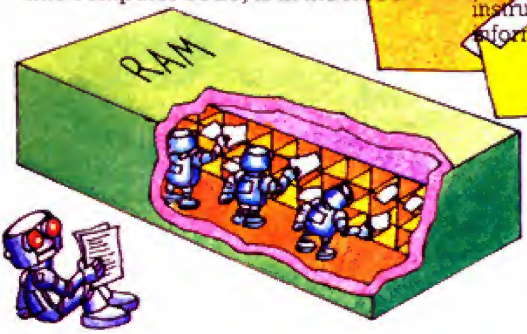


ROM stands for "read only memory". The machine code instructions which tell the computer what to do are stored in the ROM chips. It is called a read only memory because the computer can only read the information in ROM, it cannot store new information there. On most home computers, the interpreter (the program which translates BASIC into computer code) is in the ROM.

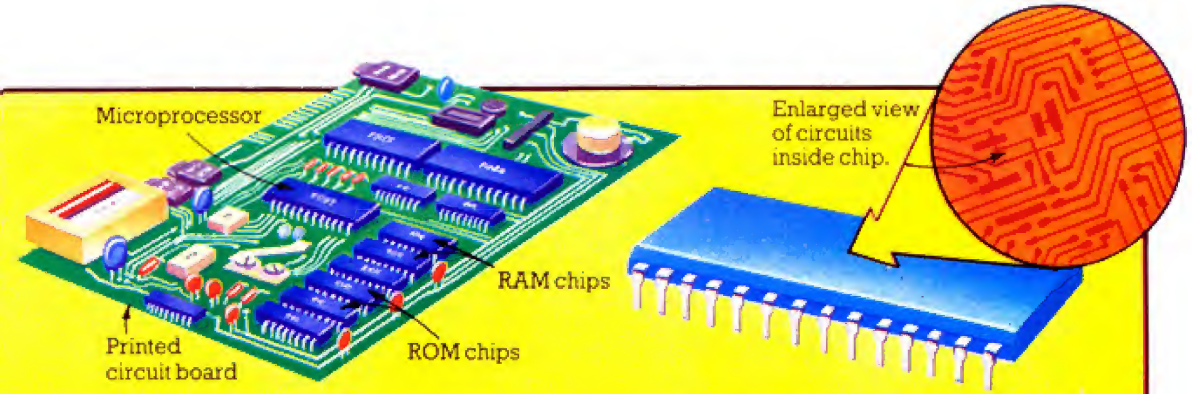


Bytes of computer code flow between the chips along the tracks of the printed circuit board. There are three separate systems of tracks for carrying bytes for doing different jobs. Each system of tracks is called a "bus".

The RAM chips



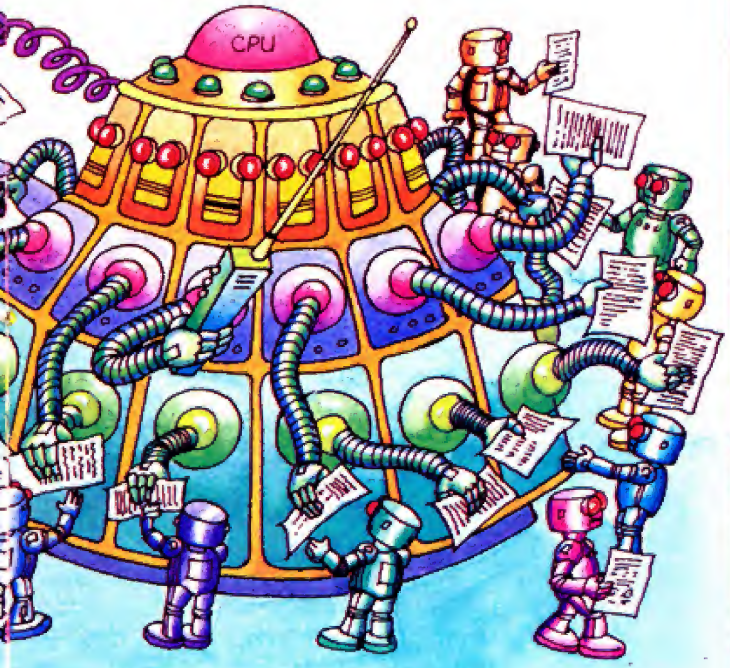
RAM stands for "random access memory". This is where the programs you give the computer are stored while the computer is working on them. It is called a random access memory because the computer can find, or access, any piece of information anywhere in the memory. When you switch the computer off the information stored in RAM is wiped out.



The proper name for a chip is an "integrated circuit" and inside each chip there are microscopic electrical circuits. All the computer's work is done by streams of pulses representing instructions in binary code, flowing through the circuits in the chips. There are different chips for carrying out different tasks. The work done by the different kinds of chips is shown in the picture below.

Clock

This is a quartz crystal which pulses millions of times a second and regulates the flow of pulses inside the computer.



The microprocessor

The microprocessor chip holds the computer's central processing unit, or CPU. This is where all the computer's work is done. The CPU does calculations, compares pieces of data, makes decisions and also co-ordinates all the other activities inside the computer. The information telling the CPU what to do is in the ROM.

Your computer's microprocessor

There are several different makes of microprocessor chip and all home computers use one or other of them. The most common makes are the Z80, found in Sinclair (Timex) computers, and the 6502, used in the Oric, the BBC micro, the VIC 20 and many others. Other examples of microprocessors are the 6809 used in the Dragon computer and the 9900 in the TI-99/4.

The different makes of microprocessor use different machine code instructions, but all computers with the same kind of microprocessor use the same machine code. There are also several different versions of each chip. For example, the Z80A is a faster version of the Z80 and the 6502A and 6510 (used in the Commodore 64) are versions of the 6502. They use the same machine code instructions, though, as the chips from which they were developed. This book is written specially for computers with microprocessors which understand Z80 or 6502 machine code.

The computer's memory

The easiest way to think of the computer's memory is as lots of little boxes, each of which can hold one byte, i.e. one instruction or piece of information in machine code. Each box in the memory is called a "location", and each location has a number, called its "address", so the computer can find any box in the memory.

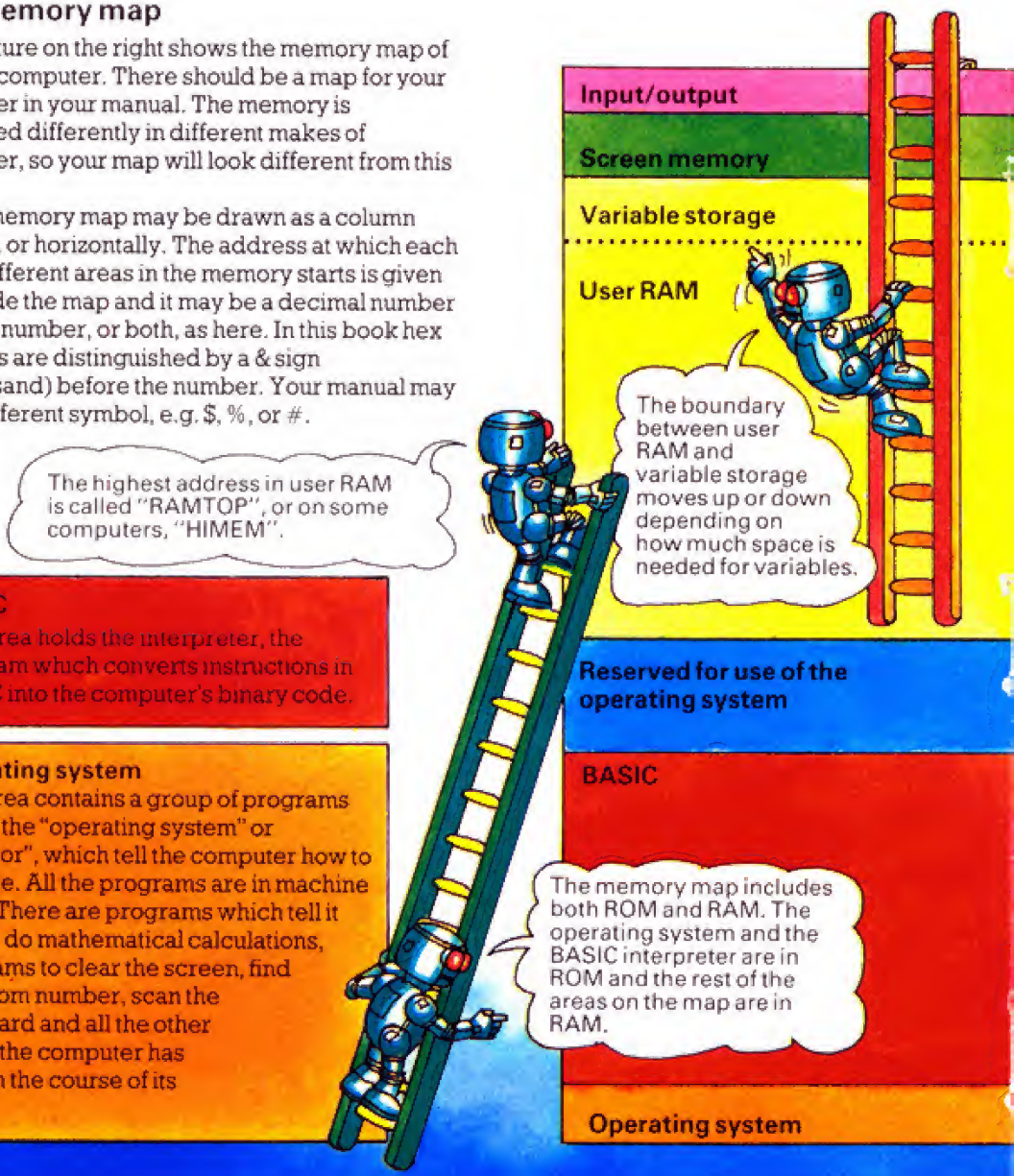
Different areas of the memory are used for storing information for different tasks and a chart giving the address where each area starts is called a "memory map".

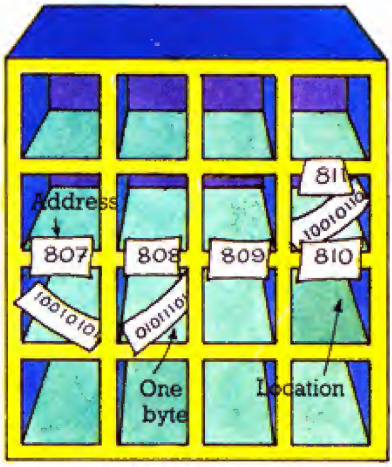
When you are programming in machine code you have to tell the computer where to find or store each instruction or piece of information. You do this by giving it the address of a memory location. You even have to tell it where to store the machine code program itself, so you need to get to know the memory map of your computer.

The memory map

The picture on the right shows the memory map of a home computer. There should be a map for your computer in your manual. The memory is organized differently in different makes of computer, so your map will look different from this one.

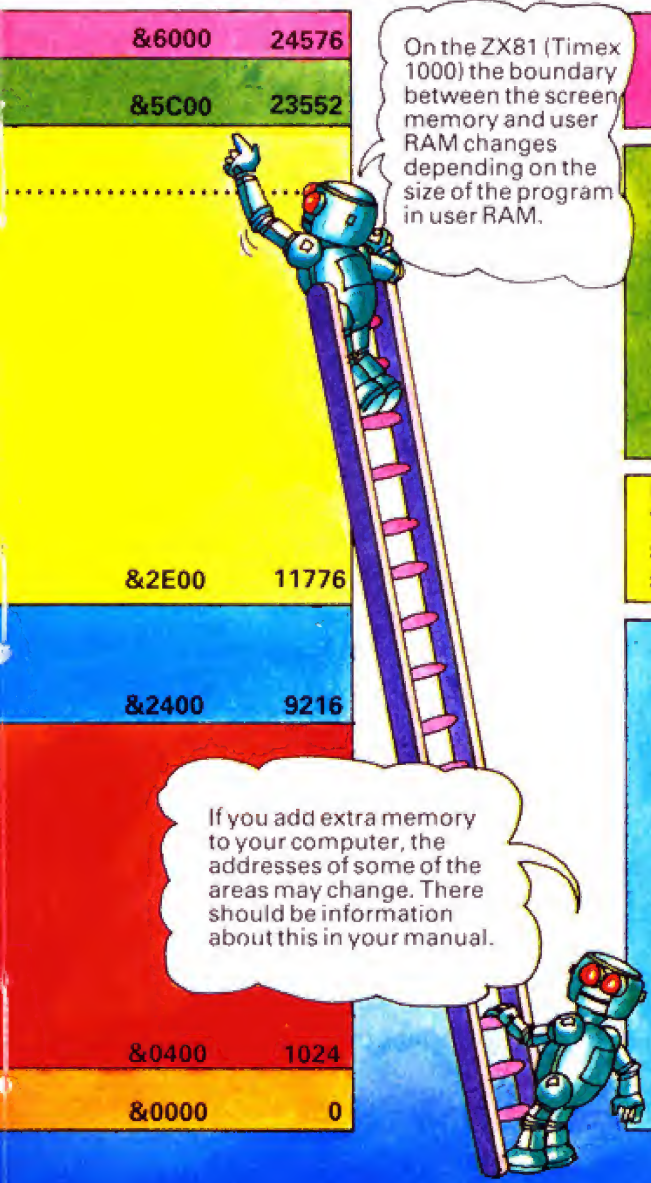
The memory map may be drawn as a column like this, or horizontally. The address at which each of the different areas in the memory starts is given alongside the map and it may be a decimal number or a hex number, or both, as here. In this book hex numbers are distinguished by a & sign (ampersand) before the number. Your manual may use a different symbol, e.g. $, %, or #.





Memory addresses

Inside the computer, memory addresses are represented by two bytes of computer code, i.e. 16 pulse or no-pulse signals or "bits". The largest possible memory you can have on a microcomputer which uses a Z80 or 6502 microprocessor is 64K (ROM and RAM combined). This is because the biggest number you can make with 16 binary digits is 65535, so this is the highest possible address. This gives 65536 locations, numbered from 0 to 65535. Each location holds one byte, 1024 bytes make a kilobyte (K) and 65536 bytes equal 64K (65536 ÷ 1024 = 64).



Input/output
These memory locations are linked to the input/output sockets on the computer.

Screen memory
Sometimes called the "display file", this part of the memory holds information displayed on the screen. Any information stored in the screen memory is automatically shown on the screen. Most microcomputers have a "memory mapped display" in which each location in the screen memory holds the information for one particular position on the screen.

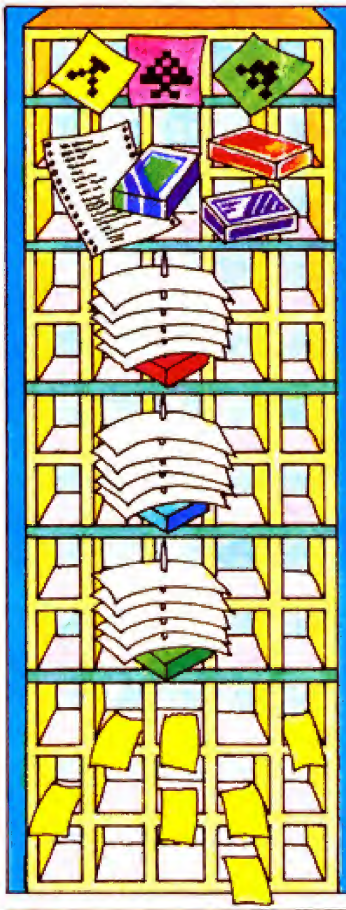
User RAM
This is where the programs you type in are stored. The data for variables and arrays is stored at the top of user RAM.

Reserved for use of the operating system

These RAM locations are used by the computer to keep track of everything going on while it carries out a program. For instance, information about the position of the cursor, the current screen colour, which key is being pressed and the current program line number are all stored in this area. It is divided up into smaller areas for carrying out different tasks. Some computers have a second map of this area. You can find out more about it over the page.

Inside the computer's workspace

This picture gives a closer view of the area of the computer's memory reserved for use by the operating system. There may be a second detailed map of this area in your manual, or a list of the various addresses and what they are used for. On some computers (e.g. Sinclair/Timex), the locations used by the operating system are not in one group and are distributed throughout the memory.



User-defined graphics
If you make up your own graphics characters they are stored here.

Buffers
These are temporary stores to hold data coming in from the keyboard, or being sent to a printer or cassette.

Machine stack
Also called the processor stack, the CPU uses these locations to store addresses while it is working on a machine code program.

BASIC stack
Also called the GOSUB stack, this is for storing the line numbers used in BASIC GOSUB and GOTO commands.

Calculator stack
This is the CPU's temporary store for numbers used in calculations.

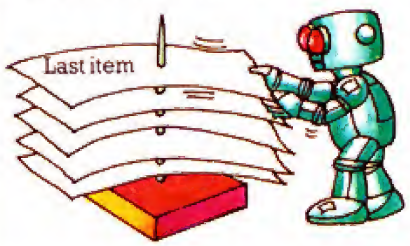
Systems variables
These are a series of memory locations where the CPU stores information about what is happening inside the computer. For instance, there are separate locations for recording the current position of the cursor on the screen, which key is being pressed and the address of the area where variables are stored.

Memory pages

To help the computer find its way around, the memory is subdivided into "pages". On a microcomputer, one page is 256 locations and four pages make one kilobyte ($4 \times 256 = 1024$).

Locations 0-255 are sometimes referred to as page zero. Different areas of the computer's memory often start at the beginning of a new page. For example, on the memory map on the previous page, user RAM starts on page 45, counting the first page as page zero.

More about stacks



The computer uses the stacks to store temporary data in a particular way. The last item to be stored must always be the first to be retrieved. This is called LIFO storage: last in, first out.

Hex numbers

In a machine code program, numbers and addresses are always written in hex. Below you can find out how to convert decimal numbers to hex, and vice versa.

Decimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Hex	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

This chart shows the hex digits (0-9 and A-F) and their decimal values. To make numbers over 15 (F) you use two (or more) digits, just as you do in the decimal system to write numbers over 9. The value of each digit depends on its position in the number.

Decimal			
1000s	100s	10s	1s
1	2	2	6

Hex		
256s	16s	1s
4	C	A

In the decimal system the first digit on the right of a number shows how many 1s there are, the second shows the number of 10s, the third, the number of 100s (10^2), etc.

In a hex number the first digit on the right also shows the number of 1s but the next digit shows the number of 16s, and the third digit shows the number of 256s (16^2).

Converting hex to decimal

To convert a hex number e.g. 4CA, to decimal, look up the decimal for each of the digits in the number. Then multiply each digit by the value of its position in the number and add up the answers.

256s	16s	1s	
4	C	A	
4	12	10	Decimal value
×256	×16	×1	
1024 +	192 +	10 = 1226	

&4CA is 1226 in decimal.

Can you convert &A7 to decimal and decimal 513 to hex? (Answers page 44.)

Decimal to hex

To convert a decimal number e.g. 1226, to hex, first you divide by 256 to find how many 256s there are in the number. Then you divide the remainder by 16 to find the number of 16s and the remainder from this sum gives the number of 1s. Finally, convert the answer to each sum to a hex digit.*

1226 ÷ 256 = 4 ..... 4 is 4 in hex
remainder 202

202 ÷ 16 = 12 ..... 12 is C in hex
remainder 10 ..... 10 is A in hex

1226 is 4CA in hex

Converting hex addresses

In a hex address, e.g. 5C64, the two left-hand digits show which page (see opposite) the location is on and the second pair of digits shows the position on the page.

Hex to decimal

Address &5C64

Page number = &5C = 92 decimal
Position on page = &64 = 100 decimal

92 × 256 = 23552 + 100
= 23652

Hex address 5C64 is 23652 decimal.

Decimal to hex

Decimal address 23652

23652 ÷ 256 = 92 (memory page number) remainder 100 (position on page)

92 ÷ 16 = 5 remainder 12 = &5C
100 ÷ 16 = 6 remainder 4 = &64

Decimal address 23652 is 5C64 in hex.

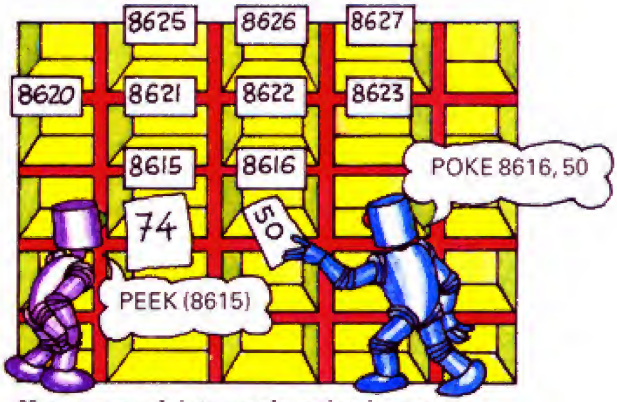
To convert a hex address to decimal, first convert each pair of digits to a decimal number, as shown above. Then multiply the page number by 256 (there are 256 locations in a page) and add the number for the position on the page.

To convert a decimal address to hex you have to divide by 256 to find the memory page number. The remainder gives the position on the page. Then you convert the figures to hex digits as described above.

*See page 41 for how to do this on a calculator.

Peeking and poking

Two BASIC words, PEEK and POKE,* enable you to look at the bytes stored in the computer's memory locations and change them. You use PEEK and POKE with the decimal, or on some computers, hex, address of a memory location. Remember, to give the computer hex numbers you must type a sign such as &, # (called hash) or $ before the number. Check this in your manual as it varies on different computers and some computers will accept only decimal numbers.



You can peek into any location in your computer's memory, but you can only poke new bytes into RAM locations because the bytes in ROM cannot be changed.

Using PEEK

```
PRINT PEEK(12345)
46
PRINT PEEK(720)
240
PRINT PEEK(8643)
0
LET A=PEEK(1024)
PRINT A
176
```

```
10 FOR J=700 TO 725
20 PRINT PEEK(J);",";
30 NEXT J
```

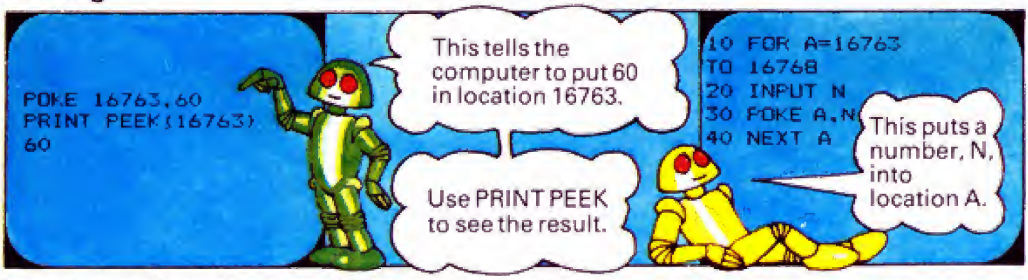
```
RUN
36,27,234,56,21,0,0,
0,0,45,32,67,121,45,
47,89,63,21,0,87,241,
202,225,63,87,16,
```

These are the decimal equivalents of bytes of computer code.

To tell the computer to look in a memory location you use PEEK (or your computer's command) with the address of that location. To see the result on the screen, use PRINT PEEK, or store the result in a variable using LET and then print out the variable, as shown above left.

Try writing a short program using a FOR/NEXT loop, like the one in the centre above, to print out the bytes from a series of locations. Look at your computer's memory map and experiment with addresses in different parts of the memory.

Poking



The picture above shows you how to use POKE. You can poke anywhere in RAM, but if you poke new values into the area reserved for use by the operating system you may disrupt the workings of the computer. You can restore it to normal by switching off and on again. Try writing a short program like the one above to poke several numbers into a series of locations in user RAM.

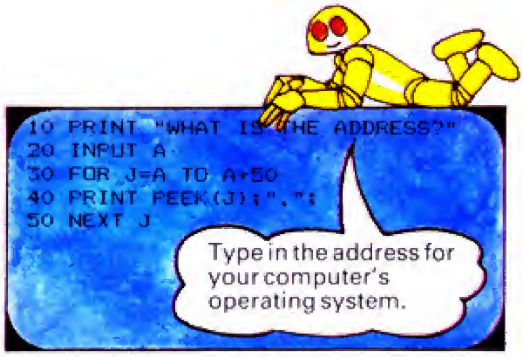
The numbers you poke must be between 0 and 255, the highest number than can be represented with eight binary digits (one byte of computer code).

*Some computers use different commands, e.g. the BBC uses a ? mark. Check your manual.

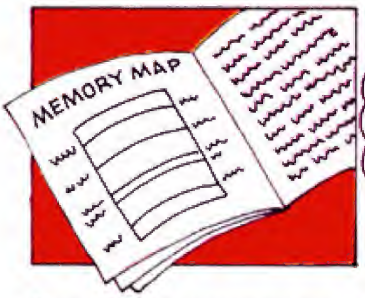
What the numbers mean

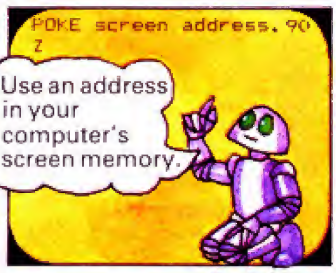
When you tell the computer to print the contents of a memory location on the screen, the result is always a decimal number from 0 to 255. This is because each memory location can hold one byte, and the highest value that can be represented with eight binary digits is 255. There are only 256 (0 to 255) possible different bytes of computer code and each byte can have several different meanings for the computer.

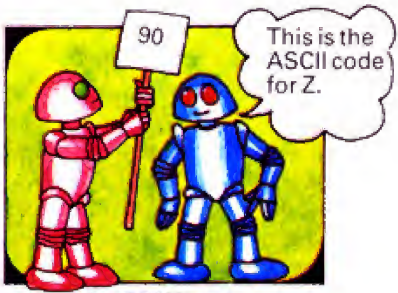
For example, the binary number 00110000 (decimal 48) could be the code for one of the instructions in the instruction set, for a letter on the keyboard, or for part of the address of another memory location (each address consists of two bytes).



Look in your manual to find the address in ROM of your computer's operating system and then try this program. The numbers which appear on the screen are the decimal equivalents of bytes of machine code from one of the programs in the operating system.

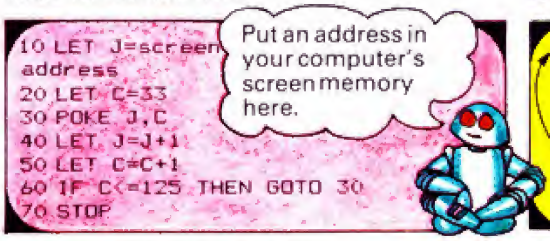


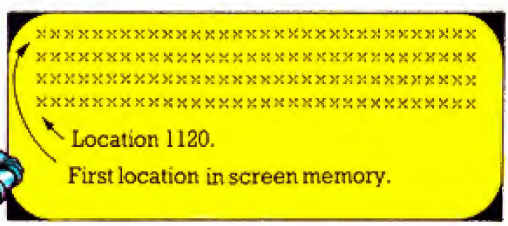




Now find the screen memory for your computer, then try poking numbers into screen memory locations. You do not need to use PRINT PEEK because bytes stored in the screen memory are automatically displayed on the screen. This time the computer interprets the number as the code for a character.*

Most computers use the ASCII code (pronounced "askey"), to decide which numbers represent which characters, but some, such as the ZX81 (Timex 1000) use different numbers. The VIC 20 has a special set of numbers, called screen codes, for characters to be displayed on the screen. There should be a list of your computer's character codes in your manual.





Try a short program like the one above to print your computer's character set. The program uses ASCII codes, starting with 33, the code for !, and ending with code 90. Other numbers in the range 0-255 are for special keys such as SPACE and DELETE, for printing the alphabet in inverse or flashing characters, and for graphics characters.

On most computers you can print a character in a particular position on the screen by working out the address of the location for that position. For example, if the screen memory starts at location 1024 and the computer can print 32 characters on a line, the address for the first position on the fourth line will be $1024+(32\times3)$ which is 1120. (Address 1024 is counted as zero.)

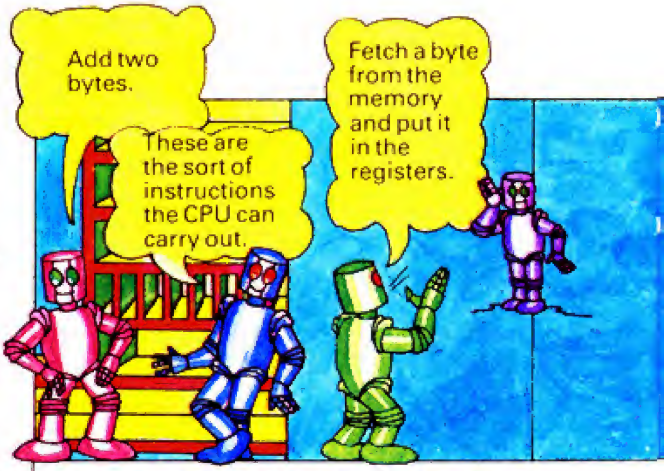
*On the Spectrum (Timex 2000) the information for each position on the screen is stored in several different memory locations and you cannot print characters by poking codes into the screen memory.

Inside the CPU

All the computer's work is done by fetching bytes of instructions and data from the memory, then carrying out the instructions in the CPU.

There are three main areas inside the CPU: the registers where bytes of data are held while they are processed; the ALU, or arithmetic/logic unit where bytes can be added, subtracted or compared; and the control unit which organizes all these activities.

The arrangement of the registers in the Z80 and 6502 chips is different, as shown in the pictures below.



These pictures show the sort of instructions which the CPU can carry out. They are all very simple. It can fetch bytes from the memory and put them in the registers, move bytes from one register to another, process them in the ALU and store the results in the memory. Even the simplest task, such as

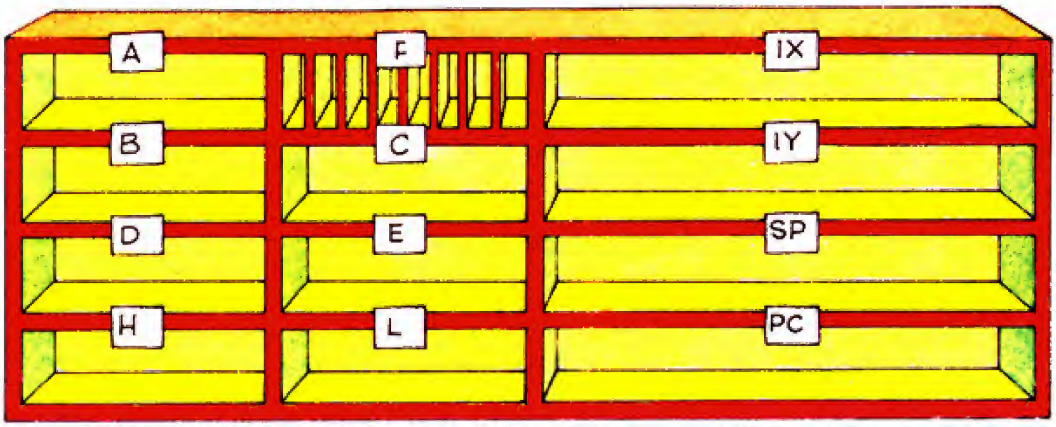
The Z80 registers

The main difference between the Z80 and the 6502 chips is that the Z80 has more registers. This means that bytes can be stored temporarily in the CPU, whereas in the 6502 they have to be sent back to the memory.

A stands for "accumulator". It is the most important register in the CPU and stores bytes on their way to and from the arithmetic/logic unit. It can only hold one byte at a time.

F is the "flags register". It holds eight bits but only six of them are used. Each bit acts as a signal. For example, the carry flag is set to 1 when an answer is greater than 255 and will not fit in one byte and the sign flag shows whether a number is positive or negative.

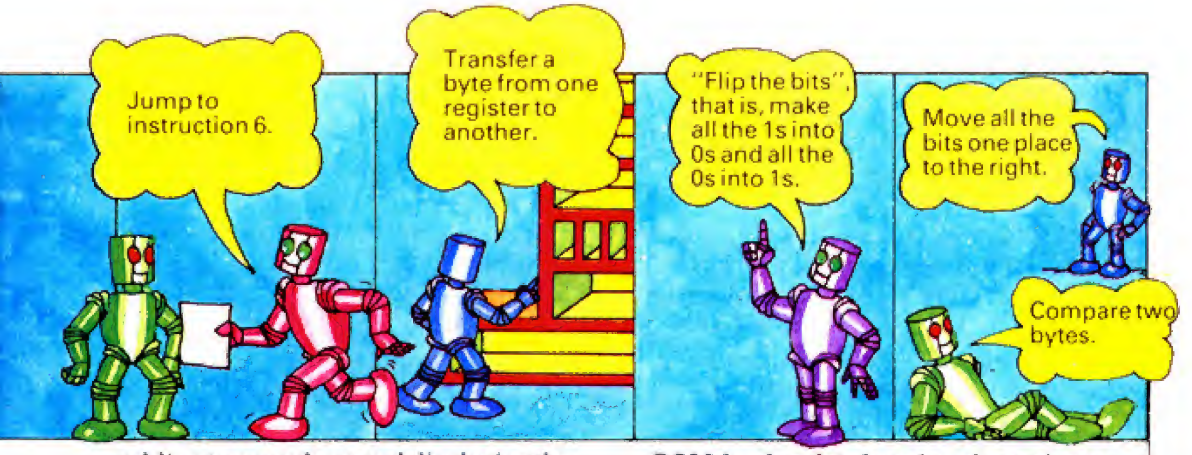
IX and **IY** are called "index registers". They can each hold 16 bits and they are used in certain instructions to work out the address of a byte in the memory.



B, C, D, E, H and **L** are general purpose registers where bytes can be stored on their way to or from the memory. Each can hold only one byte but they can be grouped together in pairs, e.g. BC, DE or HL to hold two bytes.

SP stands for "stack pointer". It is a 16-bit register and stores the address of the last item in the machine stack – the place where the CPU stores temporary data.

PC is the "program counter". It is a 16-bit register and it holds the address of the next byte to be fetched from the memory. The number in the program counter increases by one each time an instruction is carried out.



adding two numbers and displaying the result on the screen, involves over a hundred simple steps like these and the CPU can carry out over half a million each second.

For each operation the control unit fetches an instruction byte from the ROM or RAM, loads a data byte into the registers and then performs the operation specified by the instruction. In machine code, you can tell the CPU what to do with the bytes in the registers, but the ALU and control unit carry out their work automatically and you cannot tell them what to do.

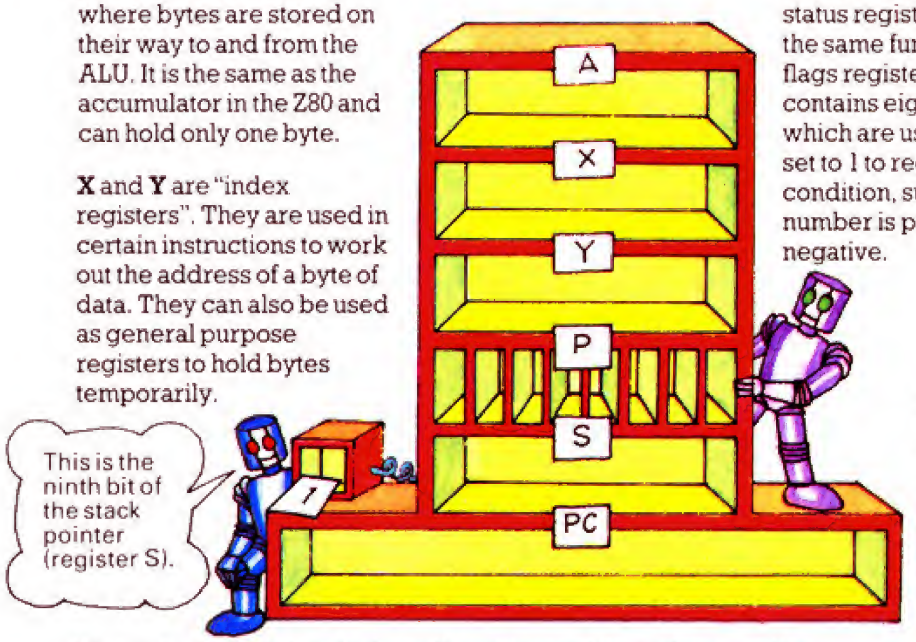
The 6502 registers

The main registers in the 6502 are the same as those in the Z80, but some of them are called by different names.

A is the "accumulator" where bytes are stored on their way to and from the ALU. It is the same as the accumulator in the Z80 and can hold only one byte.

X and **Y** are "index registers". They are used in certain instructions to work out the address of a byte of data. They can also be used as general purpose registers to hold bytes temporarily.

P stands for "processor status register" and it has the same function as the flags register in the Z80. It contains eight bits, seven of which are used. Each bit is set to 1 to record a certain condition, such as whether a number is positive or negative.

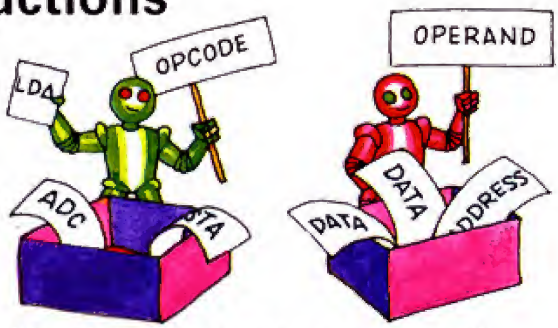


PC is the "program counter" and it works in the same way as the PC register in the Z80.

S is the "stack pointer". It stores the address of the last item on the stack – the special area in the RAM where the CPU stores data. In the 6502 the stack pointer is an eight-bit register. In order to store addresses a ninth bit kept permanently at 1 is wired up to the S register. This represents the page number of the address, so in the 6502, the stack is always in page one of the memory. The number in the stack pointer gives the position on the page.

Giving the CPU instructions

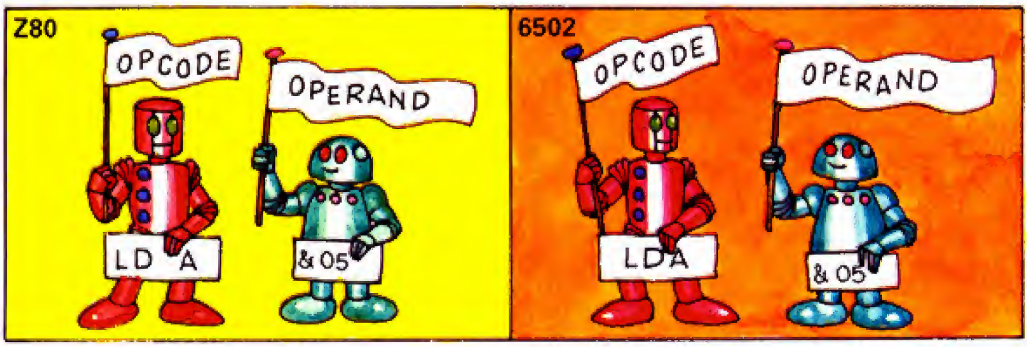
A program in machine code consists of a list of instructions telling the CPU exactly what to do with bytes in the registers. You can use only the instructions that the CPU understands, so for computers with a Z80 or Z80A microprocessor you must use instructions from the Z80 instruction set and for computers with a 6502, 6502A or 6510 microprocessor, you must use 6502 instructions. There is a list of Z80 and 6502 instructions at the back of this book.



Most machine code instructions consist of two parts: an "opcode" and an "operand". The opcode tells the CPU what to do and the operand tells it where to find the data to work on. (The word operand means "object on which an operation is performed".) Each opcode is an instruction from the instruction set.

Opcodes can be written as mnemonics – short words which represent what they do – or as the hex equivalents of the computer's binary code for each instruction. For example, LD A on the Z80 and LDA on the 6502 are the mnemonics for "load a byte into the accumulator". The same opcodes in hex are 3E for the Z80 and A9 for the 6502.

Mnemonics are much easier to understand then hex, but you cannot type them into your computer unless you have an assembler (a program which translates the mnemonics into the computer's own code).* Most people write machine code programs in mnemonics and then translate them to hex.



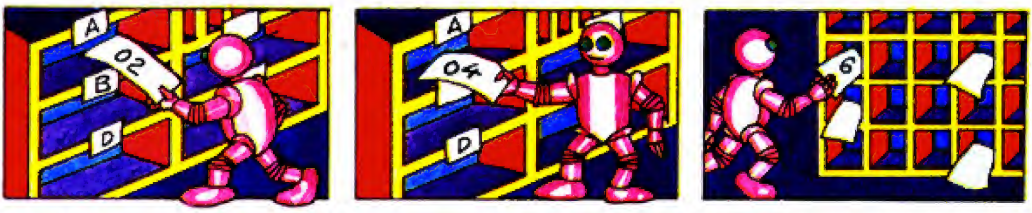
Here are two machine code instructions in mnemonics, one for the Z80 and one for the 6502. They both tell the computer to load the number 05 hex into the accumulator (& is the sign to indicate hex numbers). Numbers are always written in hex in machine code. On the 6502 a number is preceded by a # (hash) sign to show that it is a piece of data.

*You can find out about assemblers on page 40.

A simple program

Here are two programs, one for the Z80 and one for the 6502, which tell the CPU to add two numbers. They are both written in mnemonics. Strictly speaking, a program in mnemonics is called an assembly language program and one which uses hex codes is called machine code. Over the page you can find out how to translate the programs to machine code, and on the next few pages, how to load and run the version for your computer.

The Z80 and 6502 programs follow the same steps, although the actual instructions are different.* In the 6502, data on which calculations are to be carried out must always be placed in the accumulator. In the Z80 it is placed in the accumulator, or for big numbers, in register pair HL.



To add two numbers you load the first number into the accumulator. Then you add the second number to the one in the accumulator and store the result in the memory. The mnemonic opcodes for these instructions are given below.

Z80 mnemonics	Meaning
LD A, number	Load A with a number. A stands for "accumulator" and LD is short for "load".
ADD A, number	Add to A (the accumulator), a number.
LD (address), A	Load a certain address with the contents of A (the accumulator). Addresses are always written in brackets.

Opcodes and operands for the Z80 are separated by commas.

6502 mnemonics	Meaning
LDA number	Load A with a number. A stands for "accumulator" and LD is short for "load".
ADC number	ADC is the mnemonic for the instruction "add with carry". It tells the computer to add a number to the accumulator and to set the carry flag in the flags register if necessary. You can find out more about this on page 29.
STA address	Store A (i.e. the contents of the accumulator) at a certain address. ST is short for "store" and A stands for "accumulator".

Z80 adding program

```
LD A, &02
ADD A, &04
LD (&7F57), A
```

Data

Address

This program uses three opcodes: LD A, ADD A, and LD (address), A.

6502 adding program

```
LDA #&02
ADC #&04
STA &7F57
```

Data

Address

The # sign indicates that the operand is a piece of data.

Now you can fill in the data and addresses. In these examples the programs are adding 2 hex and 4 hex (which are the same as 2 and 4 decimal), and storing the result in memory location 7F57 hex.

*From now on, if you have a Z80 you can skip over the 6502 programs and if your computer uses 6502 instructions, ignore the Z80 programs.

Translating a program into hex

The only way to translate the mnemonics into hex codes is to look up each mnemonic in a chart. There is a chart of mnemonics and hex codes at the back of this book. You have to be careful, though, as there are several different hex codes for each instruction depending on whether the operand is a piece of data, an address or the name of a register. For example, here are some different versions of the opcodes for loading the accumulator, and their hex codes.

Z80		6502	
Mnemonics	**Hex codes**	**Mnemonics**	**Hex codes**
LD A, data	3E, data	LDA data	A9 data
LD A, (address)	3A, address	LDA address	AD address

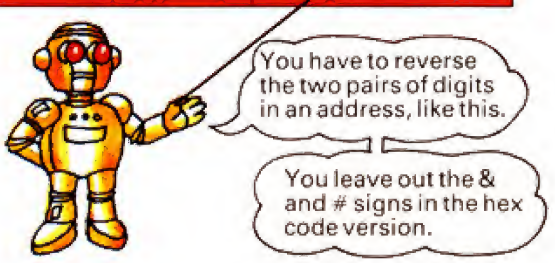
When the operand is a piece of data it is called "immediate addressing". When it is the address where the data is stored it is called "absolute addressing". The list of mnemonics and hex codes at the back of this book includes all the instructions covered in this book. If you want to write more advanced programs you will need to get a complete list of Z80 or 6502 codes and there are some suggested books on page 40.

Z80 adding program		6502 adding program	
Mnemonics	**Hex codes**	**Mnemonics**	**Hex codes**
LD A, data	3E, data	LDA data	A9 data
ADD A, data	C6, data	ADC data	69 data
LD (address), A	32, address	STA address	8D address

Here are the hex codes for the Z80 and 6502 adding programs. Instructions in mnemonics are sometimes called source code and those in hex are called object code.

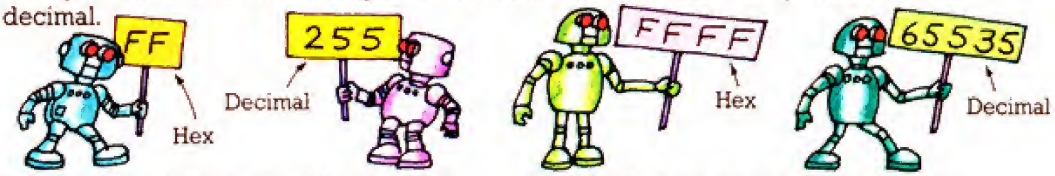
Z80 adding program with data		6502 adding program with data	
Mnemonics	**Hex codes**	**Mnemonics**	**Hex codes**
LD A, &02	3E, 02	LDA #&02	A9 02
ADD A, &04	C6, 04	ADC #&04	69 04
LD (&7F57), A	32, 577F	STA &7F57	8D 577F

Now you can fill in the data and addresses. This is quite straightforward – except for the addresses. In machine code you have to reverse the order of the two pairs of digits which make up an address. You can find out more about this on the opposite page.

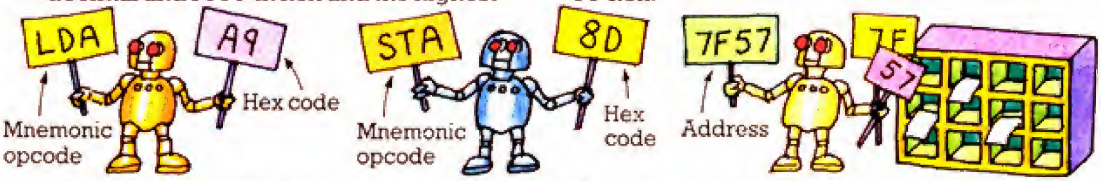


More about hex codes

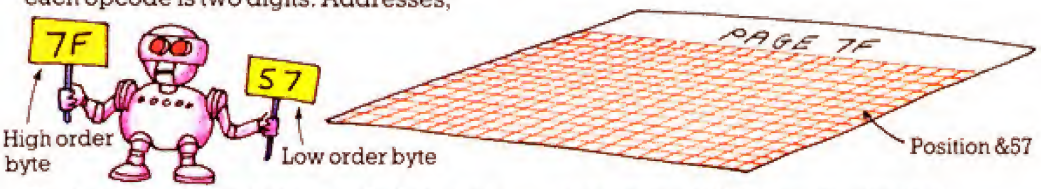
Machine code programs are written in hex rather than decimal numbers because the binary numbers used in the computer's own code translate more neatly to hex than decimal.



For example, the highest address you can have with sixteen binary digits is 65535 in decimal and FFFF in hex and the highest number that can be represented by one byte (eight binary digits) is 255 decimal and FF hex.*



Most of the opcodes in the computer's instruction set are one byte long, so in hex each opcode is two digits. Addresses, though, take up two bytes so they need two pairs of hex digits.



The first pair of hex digits is called the high order byte and it is the page number in the memory on which the address is located (see page 10). The second pair of digits is called the low order byte and it is the position of the memory location on the page (one page = 256 memory locations). Because of the way the CPU handles addresses you must always give it the low order byte (position on page) first, followed by the high order byte (page number).

Looking at machine code programs

Machine code programs in magazines look very confusing until you work out how they are presented. Below there are two examples of the way machine code listings are displayed. (Neither of these programs is complete and will not work on a computer.)

Hex dump

Each pair of digits is an instruction, piece of data or part of an address in hex.

Hex address

```
3A30  00 00 00 00 DB 01 CB 1F
3A38  CB 1F 30 07 3E 10 D3 00
3A40  CF 37 3F 21 2F 39 11 00
3A48  D0 01 C0 03 C5 0E 27 71
3A50  1A FE 5B 28 2B CD C5 3A
3A58  CD A0 3A D3 01 4E 0D 79
3A60  FE 00 FA 88 3A F5 CD BD
3A68  0B FE 64 CA 06 3B F1 71
```

This is called a hex dump. The first four digits in each line are an address and the rest of the pairs of digits are the hex codes for instructions, data and addresses. The first code in each line is stored in the address at the beginning of the line. The rest of the codes are stored in the locations following that address.

Assembly language listing

Address Hex codes Mnemonics

```
0340   A2 00       LDX #&00
0342   BD 4E 03    LDA &034E,X
0345   9D C0 83    STA &83C0,X
0348   E8          INX
0349   E0 0B       CPX #&0B
034B   D0 F5       BNE &F5
034D   00          BRK
```

This listing includes hex codes and mnemonics. The first number in each line is the address where the first byte in each line is stored in the computer's memory. The next column contains the hex codes for the program, followed by the mnemonics.

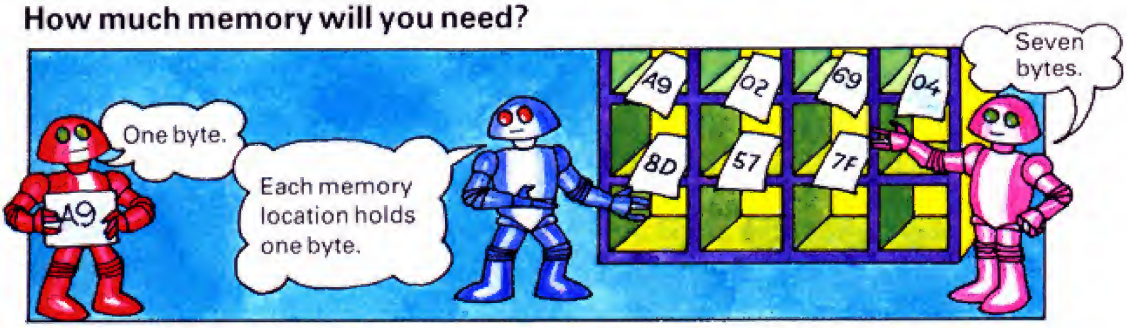
*You can find out how to convert binary numbers to decimal on page 28.

Finding free RAM

There are several things to do before you can load and run the adding program on page 18. First you need to choose an area in the memory in which to store the program. When you type in a BASIC program, the BASIC interpreter automatically stores your program in user RAM. When you give the computer a machine code program, you bypass the interpreter so you have to tell the computer where to store the program.

You need to choose an area in the RAM where your machine code will not interfere with any other information stored in the memory. For instance, you must not store machine code in the areas reserved for use by the operating system, such as the systems variables or the stacks. If you do the system will probably crash as your machine code will have replaced vital information which the computer needs to organize all its work. You also have to be careful to keep your machine code separate from any BASIC program you may give the computer at the same time. If the computer crashes the only way to restore it is to switch it off and on again and you will lose your program.

How much memory will you need?



It is quite easy to work out the length of a machine code program – you just count up the number of pairs of hex digits (each pair takes up one byte). For example, the adding program has seven bytes.

Most machine code programs are quite short and to start with a hundred bytes of memory space will probably be plenty for your machine code programs.

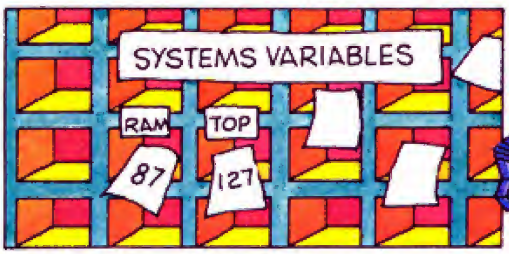
Finding free RAM

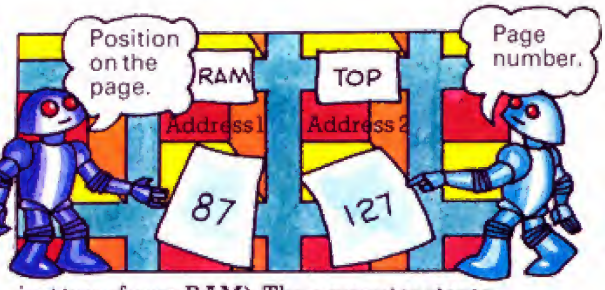
The normal place to store machine code programs is at the top of user RAM, the place where BASIC programs are stored. You have to make sure, though, that the machine code will not get mixed up with any BASIC programs. To avoid this you can lower the top of the user RAM area. This makes a "no-man's land" above user RAM which the computer will not use until you tell it to when you load your machine code program.

The top of user RAM is called RAMTOP, or HIMEM, or just top of memory. You can find out how to lower RAMTOP on the opposite page.

Lowering the top of user RAM

The computer keeps a record of the address of RAMTOP in the systems variables and you can change RAMTOP by changing the address stored in the systems variables. The instructions for doing this vary on different computers, but most follow the principles given below. You should check how to change the top of RAM in your manual though, as your computer may use different instructions, or may even have an easier way to make space for machine code.

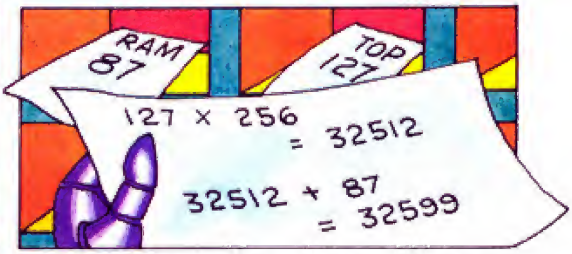




The address of RAMTOP takes up two consecutive locations in the systems variables, one for the page number of the location and one for the position on the page. Look up the addresses of these systems variables locations in your manual (they may be listed as RAMTOP, HIMEM, or just top of user RAM). The computer stores the two bytes of the address in reverse order – first the position on the page, then the page number, so the first location in the systems variables holds the position number and the second, the page.

```
PRINT PEEK(address 1)+PEEK
(address 2)*256
```

You can use PRINT PEEK (or your computer's command) like this to peek into the systems variables and print out the address of RAMTOP. Fill in the addresses of your systems variables.

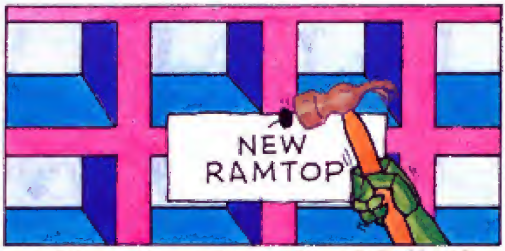


This command automatically converts the two bytes of the RAMTOP address into a decimal address by multiplying the page number by 256, then adding the position on the page.

```
CLEAR ramtop address - 100
                        Spectrum

HIMEM ramtop address - 100
                        Oric
```

Most computers have their own special command for changing the address of the top of user RAM. For instance, for the Spectrum (Timex 2000) the command is CLEAR and for the Oric it is HIMEM. These commands are followed by the address of the top of user RAM minus the number of bytes of memory you wish to reserve for machine code as shown above left. Check your computer's command in your manual.

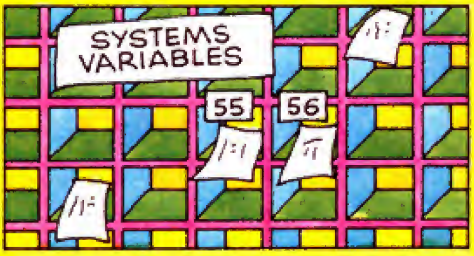


These commands lower the top of user RAM by 100 locations and so reserve an area of 99 bytes for machine code starting at the address after RAMTOP. You can change the figure 100 to reserve more space.

*See over the page for how to lower the top of RAM on the VIC 20, and where to store machine code on the ZX81 (Timex 1000).

VIC 20 tip

The VIC 20 has no special command for changing the address stored in the systems variables. Here are the instructions for lowering the address of the top of user RAM on the VIC.



The address is held in systems variables 55 and 56. Remember, the second location holds the page number.

```
POKE 56,PEEK(56)-1
```

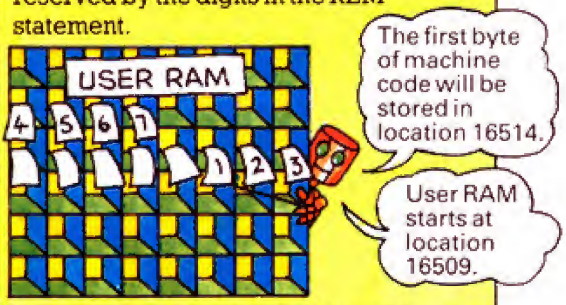
To lower the top of user RAM by 256 locations, i.e. one page, use the direct command shown above. This makes the computer peek into location 56 (the one which holds the page number). It subtracts 1 from the value held there and then pokes the new value back into location 56. In other words, it reduces the page number part of the address by 1. To see the new address of the top of user RAM type this command:
PRINT PEEK(55)+PEEK(56)*256.

ZX81 tip

On the ZX81 the best place to store machine code programs is at the beginning of user RAM. To do this you type a REM statement as the first line of the hex loader program given on page 24 and fill it with as many digits as there are bytes in your machine code program.

```
5 REM 1234567
      ← Seven bytes
```

Each of the digits in the REM statement takes up one location in the memory. Now you can poke your bytes of machine code into the locations reserved by the digits in the REM statement.

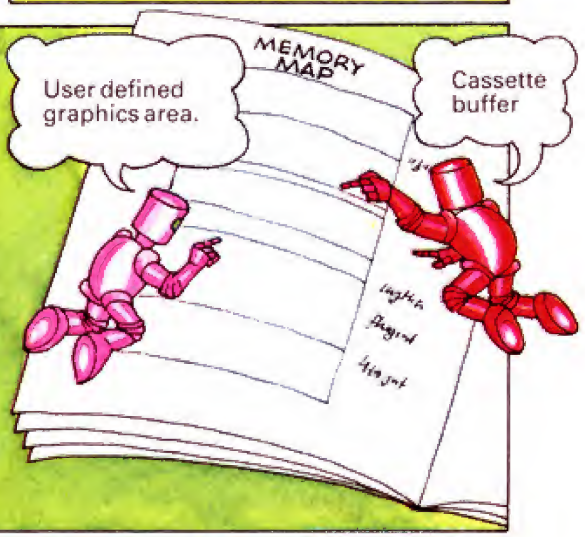


To do this you need to know the address where the first digit is stored. User RAM starts at location 16509 and the computer needs two bytes to hold the REM line number, one for REM, one for NEWLINE and one to record the length of the line, so the first digit is in location 16514.

Other places to store machine code

There are a few other places in the memory where you can store machine code, if you are not using them. For instance, if you are not planning on saving your program, you can store it in the cassette buffer, or if you are not creating any user-defined graphics, you could store it in the area set aside for this. Look in your manual to find the addresses of these areas in the RAM.

Your manual may also suggest suitable places in your computer's memory for storing machine code. You should look out, too, for tips in magazines and books.

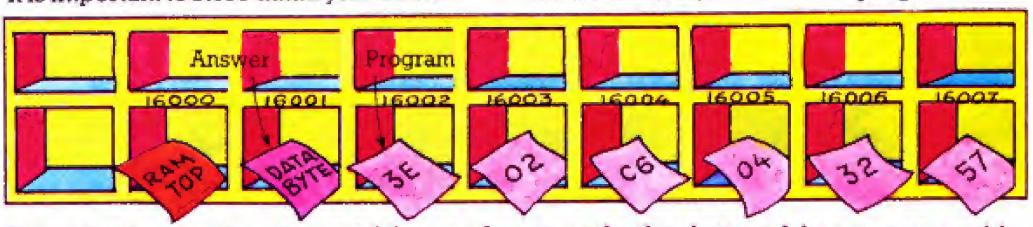


Loading and running a program

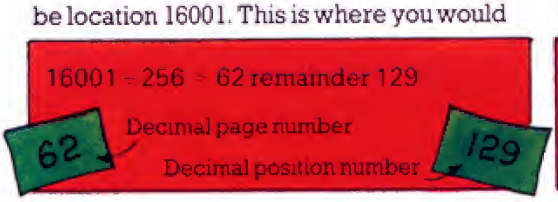
The next few pages show you how to load and run the adding program on page 18. To give the computer a machine code program you have to poke each byte into the area of memory that you have chosen for storing machine code (e.g. above RAMTOP). On most computers you can only poke decimal numbers so you use a short BASIC program called a "hex loader" to do this for you. The hex loader converts each byte of machine code to a decimal number, then pokes it into the memory. There is a hex loader program over the page. First, though, you need to change the address for the answer to the adding program, to an address suitable for your computer. There is also one more instruction (see below) you must add to the program.

Choosing an address for the answer

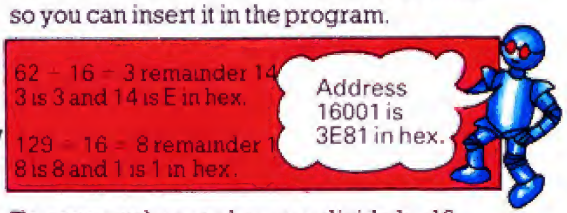
Data produced by a machine code program, such as the answer to the sum in the adding program, is called "data bytes". It is important to store data bytes where they will not get mixed up with the program itself. The best place is right at the beginning of the area you have reserved for machine code, in front of the program.



For example, if you have lowered the top of user RAM to, say, location 16000, the first address of the area for machine code will be location 16001. This is where you would store the data byte and the program would start in location 16002. You will need to convert the address for the data byte to hex so you can insert it in the program.

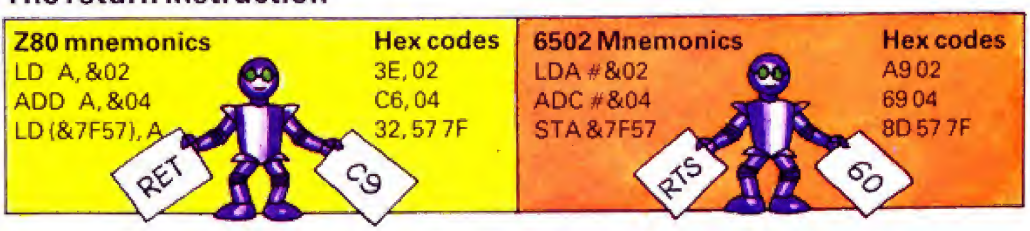


To convert the address to hex you divide by 256. The answer is the decimal page number and the remainder is the position on the page (see page 11).



To convert these to hex you divide by 16 and then convert the answers and remainders to hex digits as shown above.

The return instruction



At the end of every machine code program you must always have the instruction RET (for the Z80) or RTS (for the 6502). This makes the computer stop running the machine code program and return to where it left off. Without this command, the computer would carry on attempting to follow an instruction for every byte it found in the memory and the system would soon crash.*

*There is more about the return instruction on page 35.

Hex loader program

Here is the program for loading machine code into the computer's memory. To use this loader you put the hex codes of your machine code program in line 160, followed by the word END, as a signal to tell the computer there is no more data. At line 40, the computer reads a pair of hex digits, converts them to a decimal number in lines 70 to 110 and then pokes that number into the memory in line 130.*

```
10 PRINT "ADDRESS WHERE MACHINE
CODE IS TO BE STORED?"
20 INPUT A
30 LET C=0
40 READ H$
50 IF H$="END" THEN GOTO 180
60 IF LEN(H$)<>2 THEN GOTO 170
70 LET X=(ASC(H$)-48)*16
80 IF ASC(H$)>57 THEN LET X=
(ASC(H$)-55)*16
90 LET Y=ASC(RIGHT$(H$,1))
100 IF Y<58 THEN LET X=X+Y-48
110 IF Y>57 THEN LET X=X+Y-55
120 IF X<0 OR X>255 THEN GOTO 170
130 POKE A+C,X
140 LET C=C+1
150 GOTO 40
155 REM SAMPLE DATA ONLY
160 DATA EF,F6,E2,A9,END
170 PRINT "BAD DATA"
180 STOP
```

A is the address of the first location where you wish to store your program.

C is a counter.

Puts first pair of hex digits in line 160 into H$.

Tests H$ for word END, the signal to indicate end of data.

Checks to make sure H$ contains two digits, and if not, goes to line 170.

Converts first hex digit to a decimal number and stores in X.

Converts second hex digit to a decimal number, Y, and adds to X.

Checks for bad data by making sure decimal number in X is between 0 and 255.

First time, C=0, so pokes X into memory location A.

Adds one to C, so pokes decimal value of next hex code into memory location A+1.

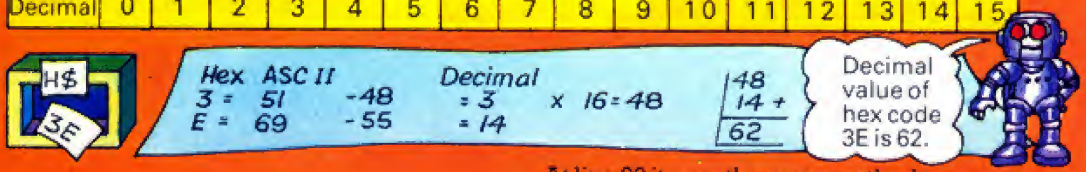
Back to read next hex code.

Put your hex codes here, followed by signal word END.

Prints this if it finds bad data in lines 60 or 120, then stops.

How the loader works

Hex	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
ASCII	48	49	50	51	52	53	54	55	56	57	65	66	67	68	69	70
	minus 48										minus 55					
Decimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15



At line 70, the computer converts the first digit in H$ to its ASCII code using the BASIC word ASC. It then converts the ASCII code to a decimal value by subtracting 48, or for codes over 57, by subtracting 55, as shown in the chart above. Then it multiples by 16 because the first hex digit represents the number of 16s and puts the answer in X.

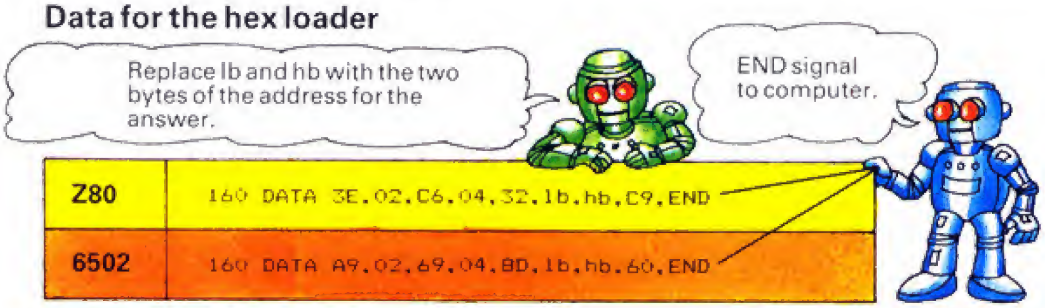
At line 90 it uses the same method to convert the right-hand digit to an ASCII code and stores it in Y. In lines 100 and 110 it changes Y to a decimal number by subtracting 48 or 55 as before, and adds it to X. (This time it does not multiply by 16 as it is the digit which represents 1s in the hex number.) The value stored in X is the decimal equivalent of the pair of hex digits.

*For the Spectrum (Timex 2000) change the ASC command to CODE and put each pair of hex codes in quotes. See page 48 for alterations for the ZX81 (Timex 1000) and Atari computers.

Using the loader

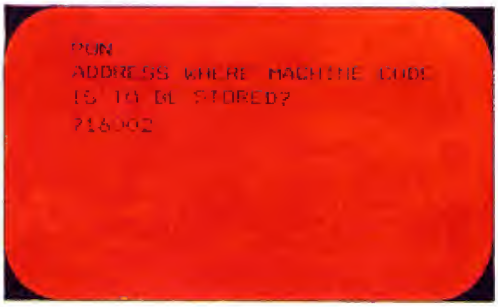
Now you can use the hex loader to try out the machine code adding program. This is not a very exciting program, but it is simple and it shows you how machine code works. Type the hex loader into your computer. At line 160, replace the sample data with the hex codes for the adding program, as shown below.

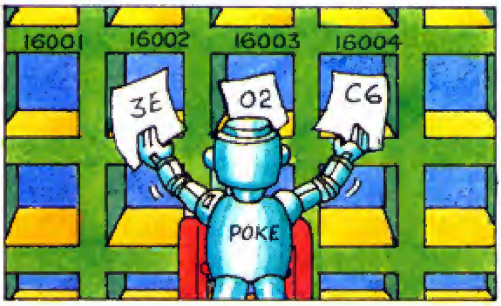
Data for the hex loader



These are the hex codes for the adding program. You need to replace the letters lb (low order byte) and hb (high order byte), with the two bytes of the address where the answer will be stored in your computer. Remember to put the bytes in reverse order, i.e. low order byte (position on page) followed by high order byte (page number).

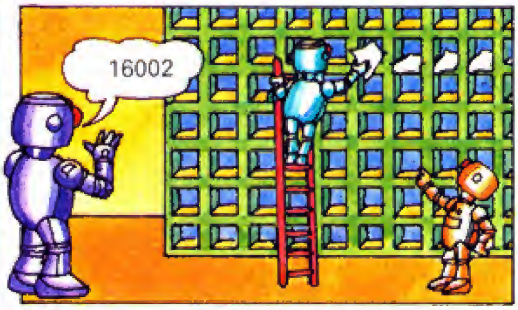
Running the hex loader





Now type RUN to run the hex loader program. When it asks you for the address, type in the first location after the one where you are storing the answer. Type this address as a decimal number as it will be used with the POKE command.

Running the machine code program



The command to tell the computer to start running a machine code program varies on different computers. Some use CALL, others use PRINT USR or SYS with the decimal address of the location where the first byte of the program is stored. Check this command in your manual. When the computer receives this command it goes to the address and starts carrying out the machine code instructions.

Seeing the result

```
PRINT PEEK(16001)
```

The computer carries out the machine code instructions and stores the answer in the location you told it to. To see the result you have to use PRINT PEEK with the address of the answer. The result will be the answer in decimal.

```
PRINT PEEK(16001)
6
```

Programs to write

You now know enough machine code to write some simple programs. There is a checklist at the bottom of the page to help you remember all the things you have to do when you write a machine code program. Answers page 44.

1. Try writing a program to add 25 and 73 (decimal) and store the result in the memory.
2. See if you can write a program to add 64 and 12 and 14 (decimal) and store the result in the memory.

The adding program will only add numbers which total less than 255. On page 28 you can find out how to add larger numbers.

Machine code checklist

1. Write your program in assembly language and convert any data to hex.
2. Look up the hex code for each of the mnemonics (there is a list of the mnemonics and hex codes at the back of the book).

Don't forget to put END after your list of hex codes in the hex loader.

3. Add the return instruction to the end of the program. (See page 23.)
4. Count up the number of bytes and reserve your free RAM area. (See pages 20-22.)

Make a note of the addresses of data bytes and of the address where you have stored the program.

5. Work out what memory locations you need for data bytes and convert the addresses to hex. (See page 23.)
6. Fill in the addresses in the program – remember to put the two bytes in reverse order. (See pages 18-19.)

Before running the hex loader, check the hex codes in the data line very carefully.

7. Type in the hex loader (you could save this program on tape) and fill in the hex codes in line 160 followed by the END signal. (See page 24.)

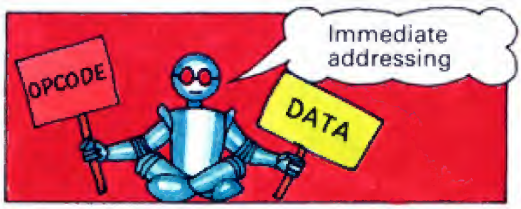
If your programs won't run, check that you have used the correct hex codes.

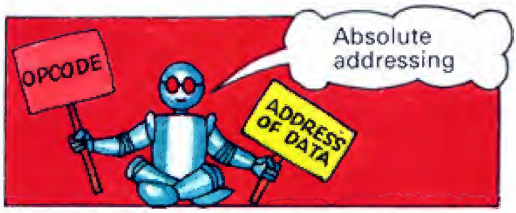
8. Run the hex loader and input the decimal address of the first location where you wish to store the machine code. (See page 25.)
9. Run the machine code program using your computer's command with the address (in decimal) of the first location where the machine code is stored. (See page 25.)

If you change the data in the hex loader you have to run the program again to poke the new bytes into the memory.

Adding bytes from memory

In the previous program the data was included in the program itself. This is called immediate addressing. Sometimes, though, you may want to tell the computer to do something with data stored in its memory. In this case, the operand part of an instruction will be an address telling the computer where to find the data. This is called absolute (or direct, or extended) addressing.





These are just two of the several different ways in which you can tell the computer where to find the data to work on. The different ways are called "addressing modes". There is a different hex code for each instruction depending on the addressing mode you are using.

Program to add numbers from the memory

Here is a program to add two numbers stored in the memory. Compare the hex codes for the instructions in this program, which uses absolute addressing, with those for the previous adding program which used immediate addressing.

Z80 program		
Mnemonics	**Hex codes**	**Meaning**
LD A, (address 1)	3A, address 1	Put the number in address 1 into the accumulator.
LD B, A	47	Put the number in the accumulator into register B.
LD A, (address 2)	3A, address 2	Put the number in address 2 into the accumulator.
ADD A,B	80	Add the number in register B to the accumulator.
LD (address 3), A	32, address 3	Store the contents of the accumulator in address 3.
RET	C9	Return

To add two numbers from memory you have to load them into the registers first. For this you can use the accumulator (A) and register B. You cannot load register B straight from the memory, though, so you have to put the first number into A and then transfer it to B.

6502 program		
Mnemonics	**Hex codes**	**Meaning**
LDA address 1	AD address 1	Put the number in address 1 into the accumulator.
ADC address 2	6D address 2	Add the number in address 2 to the accumulator.
STA address 3	8D address 3	Store the contents of the accumulator in address 3.
RTS	60	Return.

Running the program

To run this program, follow the steps given in the checklist on the opposite page. First, though, you will need to poke into the memory the two numbers to be added. You should choose memory locations at the beginning of the area you have cleared for machine code, to keep these data bytes separate from the instructions. Then convert the addresses to hex and insert them in the program. You need a third address for the answer. To see the result, type PRINT PEEK(address 3).

Working with big numbers

The programs on the previous few pages only work with numbers which add up to 255 or less. This is the highest number that you can represent with the eight bits in one register or memory location. To work with larger numbers you need to know a little more about the binary number system, and how to use the carry flag. Over the page there is a machine code program to add larger numbers.

Binary numbers

The binary number system works like hex and decimal numbers except that there are only two digits, 0 and 1. To make numbers bigger than 1 you use several digits and the value of each digit depends on its position in the number.

1	1	1	1	1	1	1	1
×128	×64	×32	×16	×8	×4	×2	×1

128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255

In a binary number, each digit has twice the value of the digit on its right. The first digit (the one on the right) shows how many ones there are in the number. The second digit shows the number of twos; the third, the number of fours; the fourth the number of eights and so on, as shown above. To convert a binary number to decimal you multiply each digit by the value of its position in the number and add up the answers.

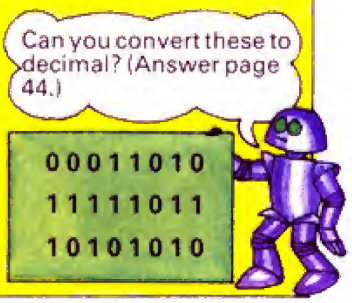
0	0	1	0	1	1	1	0
×128	×64	×32	×16	×8	×4	×2	×1

0 + 0 + 32 + 0 + 8 + 4 + 2 + 0 = 46

1	0	0	0	0	1	1	1
×128	×64	×32	×16	×8	×4	×2	×1

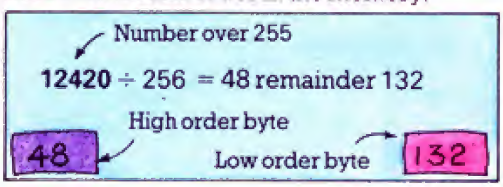
128 + 0 + 0 + 0 + 0 + 4 + 2 + 1 = 135

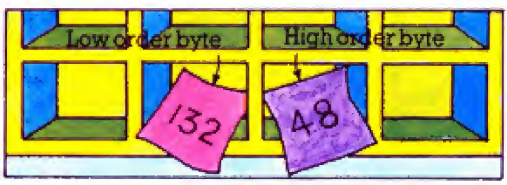
Here are some more examples which show how you convert binary numbers to decimal.



Giving the computer big numbers

Inside the computer, numbers over 255 are stored in two bytes, called the "high order byte" and the "low order byte", just like addresses. The high order byte shows how many 256s there are in the number and the low order byte is the remainder. As with addresses, the computer always deals with the low order byte before the high order byte and you have to store them in that order in the memory.

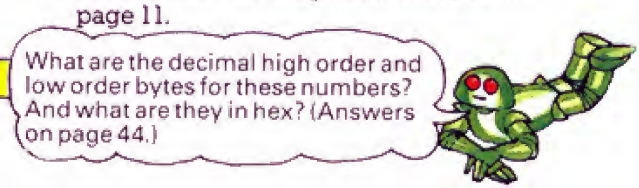




To give the computer a number over 255 you have to work out the value for each byte. To do this you divide the number by 256. The answer is the decimal value of the high order byte. The remainder is the low order byte.

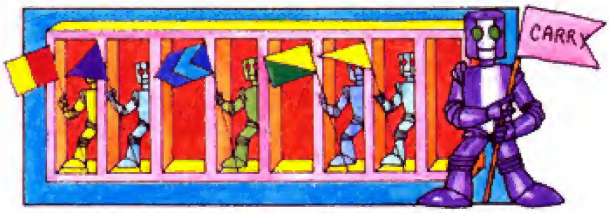
If you want to use the number as data in a machine code program you have to convert each byte to hex. To do this, divide each byte by 16, then convert the answers and remainders to hex digits as described on page 11.

307; 21214; 759; 1023.

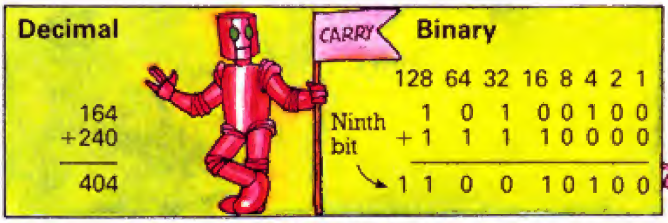


The carry flag

The carry flag is a single bit in the flags register (also called the processor status register), which is used to indicate when the answer to a sum is greater than 255 and will not fit into one byte (eight bits). Whenever this happens the computer automatically puts a 1 in the carry flag. This is called setting the carry flag and making it 0 is called clearing it.



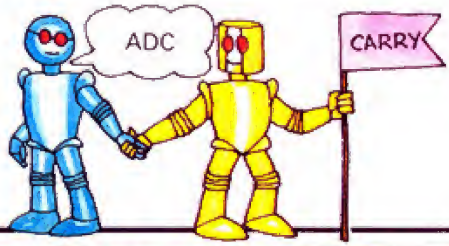
You can think of the carry flag as a ninth bit indicating that a binary 1 has been carried over from column eight of a number. For example, look at the sum 164 + 240 (10100100+11110000 in binary), below.



The answer to this sum is 404 which takes up nine bits in binary. The ninth bit shows how many 256s there are in the number. In the computer it would be represented by the bit in the carry flag.

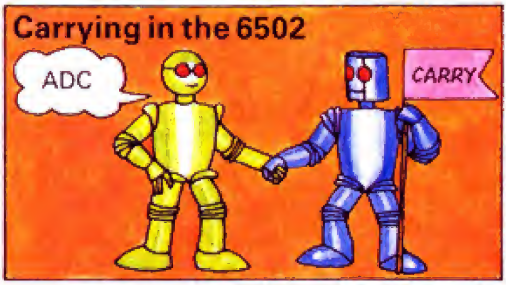
Carrying in the Z80

The Z80 has two different adding instructions: ADD and ADC. ADD tells the computer to add two numbers but to ignore any carry over from previous calculations. If the calculation results in a carry over, the computer will set the carry flag and if there is no carry it will make the carry flag 0.

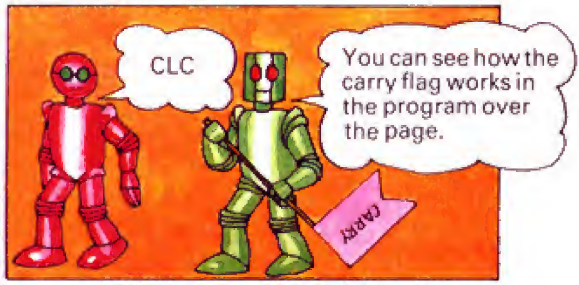


ADC stands for "add with carry" and it tells the computer to add two numbers plus the carry flag, and to set or clear the carry flag depending on the result. If you are doing a series of calculations it is best to use the ADD instruction for the first sum to make sure you do not include a carry left over from a previous operation, and then to use ADC in case there was a carry from the first calculation.

Carrying in the 6502



The 6502 has only one adding instruction, ADC, so it always includes the contents of the carry flag in calculations. Because of this it is important to clear the carry flag using the instruction CLC (clear carry flag) before you do any additions.



Big number programs

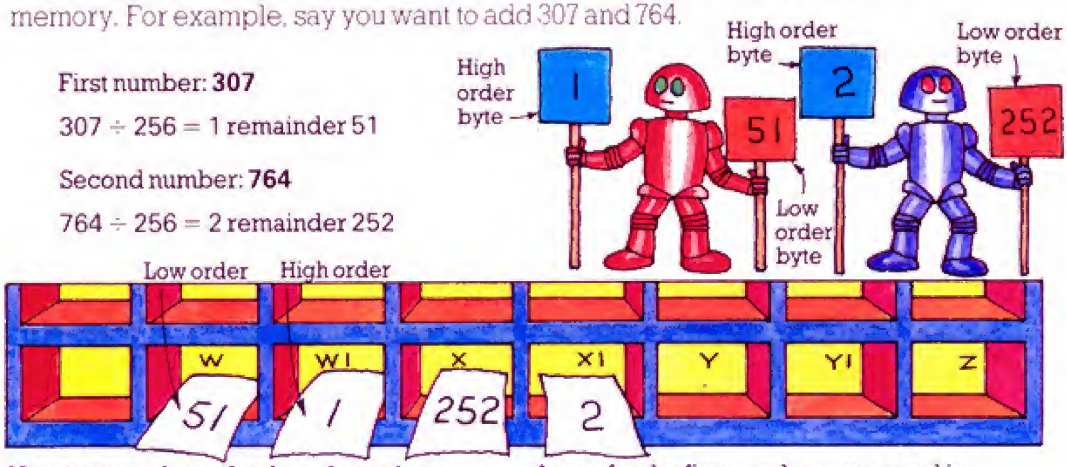
Before you can run the programs on these two pages you need to work out the high and low byte for each of the numbers you want to add and poke them into the memory. For example, say you want to add 307 and 764.

First number: **307**

307 ÷ 256 = 1 remainder 51

Second number: **764**

764 ÷ 256 = 2 remainder 252



Next you need to poke these bytes into memory locations at the beginning of the area you have reserved for machine code. For each number, the low order byte must be in the first location, followed by the high order byte. In the picture above, the two bytes for the first number are stored in locations W and W1 and the bytes for the second number are in locations X and X1. You need three locations, Y, Y1 and Z for the answer (one for the low order byte, one for the high order byte and one for a possible carry).

Z80 big number program

Adding the two numbers on the Z80 is quite easy as you can use the registers in pairs, with each pair holding the two bytes for one number. You can use the H and L registers as one pair and the B and C registers as another. When they are used like this they are referred to as HL and BC. When you are not using the accumulator you use the HL registers for adding. Here are the mnemonics and hex codes for the program. It may help you to look at the picture at the top of the page when you study this program.

Mnemonics	Hex codes	Meaning
LD HL, (address W)	2A, address W	Puts byte from address W (low order byte of first number) into register L and byte from address W1 (high order byte, first number) into register H.
LD BC, (address X)	ED 4B, address X This opcode is two bytes long.	Puts byte from address X (low order byte, second number) into register C and byte from address X1 (high order byte, second number) into register B.
ADD HL, BC	09	Adds contents of HL and BC and leaves result in HL. It does not add in the carry flag but it does set the carry flag if necessary.
LD (address Y), HL	22, address Y	Stores low order byte of answer in address Y and high order byte in address Y1.
LD A, &0 ADC A, &0 LD (address Z), A	3E,0 CE,0 32 address Z	See opposite page for how the computer checks the carry flag.
RET	C9	Return.

To run the program you need to fill in the hex addresses for W, X, Y and Z. (Don't forget to reverse the pairs of digits.) When you use the registers in pairs you need only specify one address for each pair. The computer automatically puts the byte from the next consecutive address into the other register in the pair.

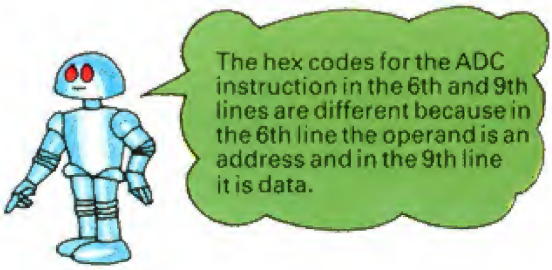
Checking the carry flag

Lines 5-7 of the Z80 program are for checking the carry flag. You cannot load the contents of the carry flag straight into a register, or into the memory. The only way to see if it has been set is to do another addition. To do this you put 0 into the accumulator (5th line), then add 0, using the add with carry instruction. If the carry flag was set by the previous calculation the accumulator will now contain 1 (from the carry flag) and this is stored in address Z (7th line).

6502 big number program

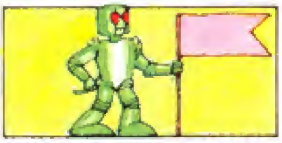
Here is the program for adding numbers greater than 255 on the 6502. Before you run it you need to work out the high order and low order bytes for the two numbers and poke them into the memory as described on the opposite page.

Mnemonics	Hex codes
CLC	18
LDA address W	AD address W
ADC address X	6D address X
STA address Y	8D address Y
LDA address W1	AD address W1
ADC address X1	6D address X1
STA address Y1	8D address Y1
LDA #&0	A9 00
ADC #&0	69 00
STA address Z	8D address Z
RTS	60

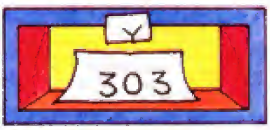


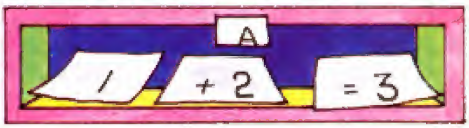
First the program clears the carry flag in case it was set by a previous operation

Then it puts the low order byte of the first number into the accumulator and adds with carry the low order byte of the second number (2nd and 3rd lines).

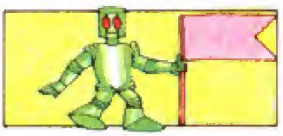


If the result is greater than 255 it sets the carry flag.





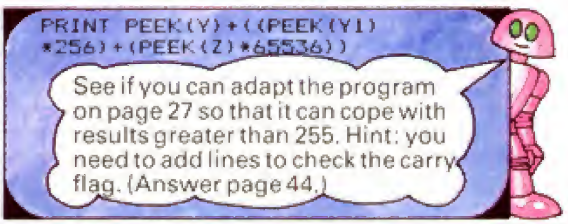
It stores the result in location Y (4th line). Then it adds the two high order bytes and the carry (if there was one) from the previous sum. It stores the result in location Y1 (7th line).



Lines 8-10 check to see if the carry flag was set using the same method as shown at the top of the page.

Seeing the result

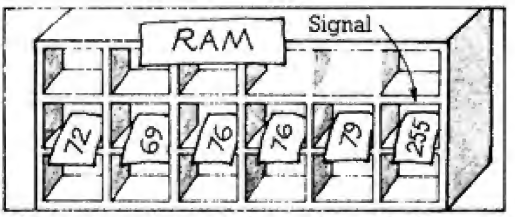
The result is stored as three bytes. The low order byte (location Y) shows the number of units. The high order byte (location Y1) shows the number of 256s. This time the carry (location Z) shows the number of 65536s. To see the result use the instruction shown on the right. (Replace Y, Y1 and Z with your computer's addresses.)



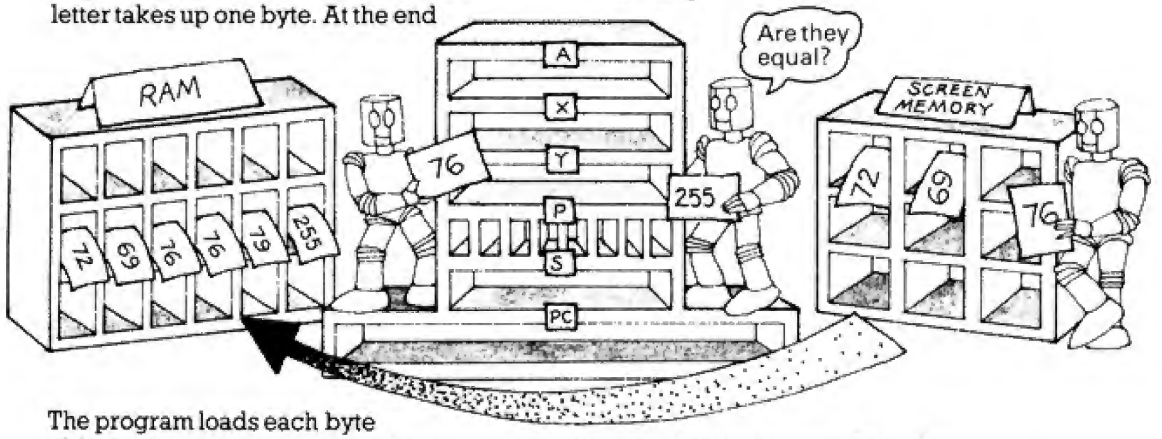
Displaying a message on the screen

The next program shows you how to use machine code to display a message on the screen. The program for the Z80 is on the opposite page and the one for the 6502 is on page 34. The two programs follow the same basic principles, although the method is slightly different for the different microprocessors.*

How the program works

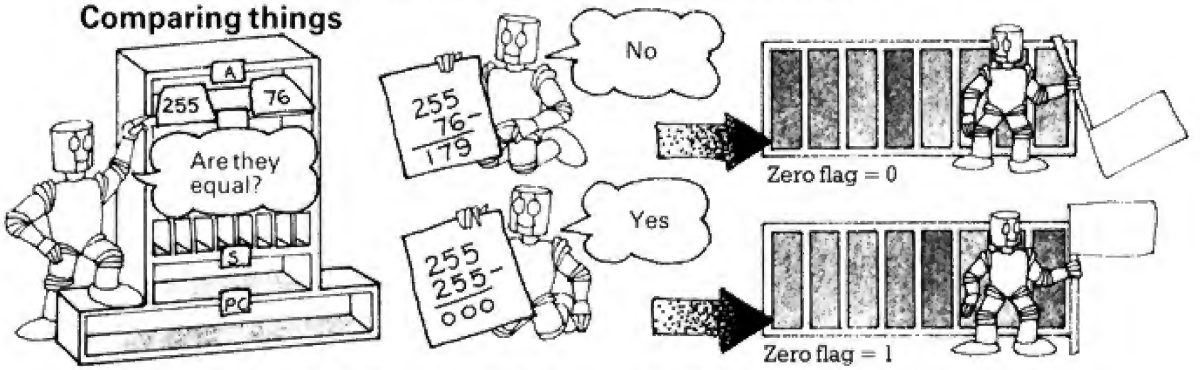


First you poke the character code for each letter of your message into locations at the beginning of your free RAM area. Each letter takes up one byte. At the end of the message you poke in the code 255 as a signal to tell the computer this is the end of the message.



The program loads each byte of the message into the accumulator and compares it with 255. If the byte of message does not equal 255, it stores it in the screen memory and it is automatically displayed on the screen. Then the computer jumps back to the beginning of the program to find the next byte of the message in the memory.

Comparing things



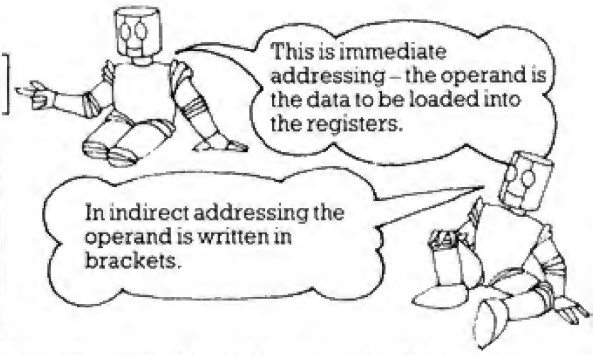
You use the opcode CP on the Z80 and CMP on the 6502 to tell the computer to compare a byte with the one in the accumulator. The computer compares them by subtracting one from the other. (This is just a test, in fact, the two bytes remain unchanged.) If the result is 0, the two bytes are equal and it sets the zero flag in the flags register to 1. If they are not equal the zero flag is 0. You can then tell the computer to go to another part of the program, or carry on with the next instruction depending on whether the zero flag is 1 or 0.

*On the Spectrum (Timex 2000) you will not get a legible message on the screen because of the way the screen memory is organized.

Z80 message program

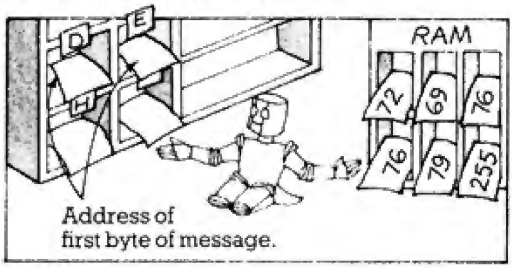
Here are the mnemonics and hex codes for the Z80. Before you run the program, poke your message into free RAM. Then fill in the addresses in lines 1 and 2 of the program. The last instruction of the program tells the computer to jump back to the third instruction.
You need to insert the address where the third instruction will be stored in your computer, into the last line of the program.

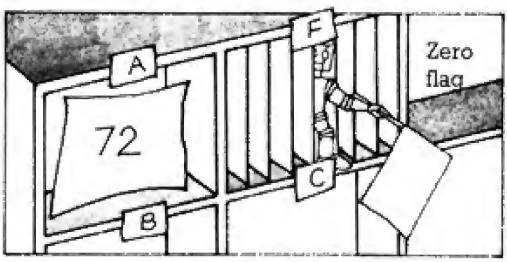
Mnemonics	Hex codes
LD HL, screen address	21, screen address
LD DE, message address	11, message address
LD A, (DE)	1A
CP,&FF	FE, FF
RET Z	C8
LD (HL),A	77
INC, DE	13
INC, HL	23
JP, address of 3rd instruction	C3, address of 3rd instruction



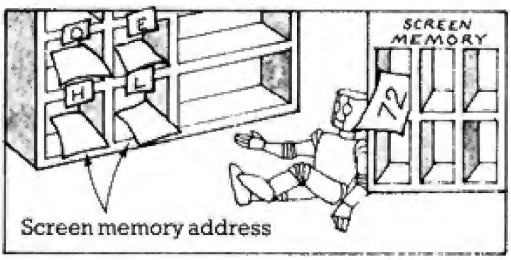
In this program, register pairs HL and DE are used as pointers to the addresses where the computer should store or fetch data. This is called "indirect addressing". The instructions in the third and sixth lines use indirect addressing.

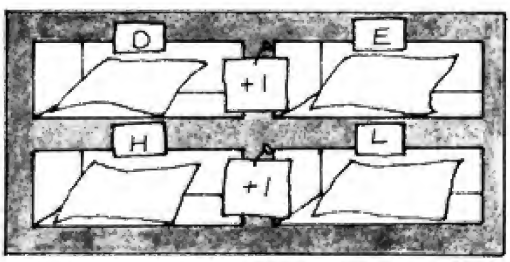
In the first two lines, the computer puts the screen address (the address where data is to be stored) into register pair HL and the message address (the address from which data is fetched), into register pair DE.





LD A, (DE) tells the computer to read the address in DE and then fetch the byte from that address and put it in the accumulator. This is indirect addressing. Then it compares the byte in the accumulator with &FF (the hex for 255). RET Z tells the computer to return to BASIC if the zero flag is 1 (i.e. if the byte equals 255). If the zero flag is 0, it carries on with the next instruction.





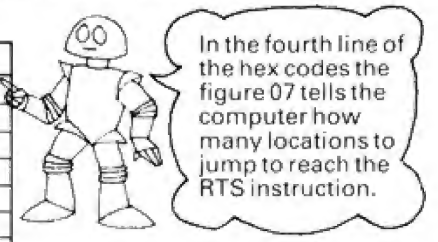
LD (HL),A also uses indirect addressing. It tells the computer to read the address in HL and then store the contents of the accumulator (the message byte) at the location with that address. INC is the mnemonic for "increment" and means increase by one. In the seventh and eighth lines the computer adds one to the addresses held in DE and HL so that when it jumps back to the instruction in the third line, it fetches the message byte from the next memory location.

6502 message program

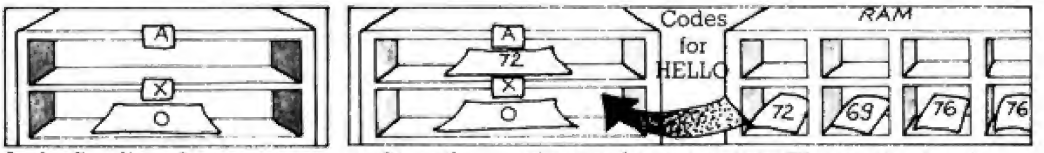
Here are the mnemonics and hex codes for the 6502. Before you run the program you need to poke the character codes for your message into free RAM, followed by 255, the signal for the end of the message. Then put the address, in hex, of the first location where the message is stored, in the second line of the program. Put an address in your computer's screen memory in the fifth line.

You also need to fill in the seventh line with the address where the second instruction in the program will be stored in your computer. This makes the computer jump back to repeat the program.

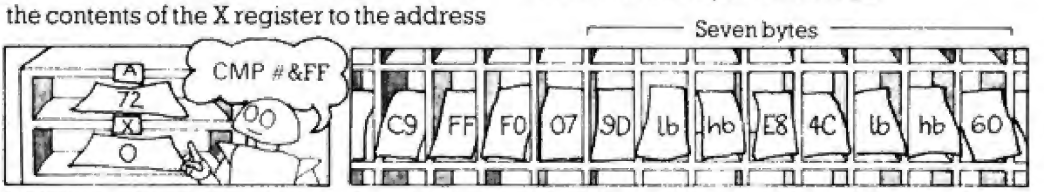
Mnemonics	Hex codes
LDX #&00	A2 00
LDA message address, X	BD message address
CMP #&FF	C9 FF
BEQ to RTS instruction	F0 07
STA screen address, X	9D screen address
INX	E8
JMP address of 2nd instruction	4C address of 2nd instruction
RTS	60



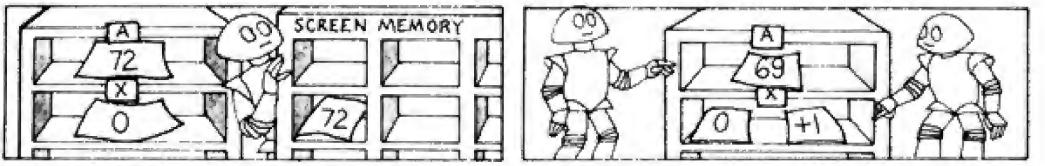
This program uses another addressing mode, called "indexed addressing". In indexed addressing, the contents of the X or Y registers are added to the operand to give the address where the data is stored. The second and fifth lines use indexed addressing.



In the first line, the computer puts 0 into the X register. The second instruction uses indexed addressing so the computer adds the contents of the X register to the address given in the instruction. The result gives it the address of the data to be loaded into the accumulator (a byte of message).



CMP in the third line makes the computer compare the byte in the accumulator with &FF (hex for 255), the signal for the end of the message. If they are equal it sets the zero flag to 1. The next instruction, BEQ, stands for "branch if equal" (i.e. if the zero flag is 1). In the hex codes it is followed by a number telling the computer how many locations to jump. We want the computer to branch to RTS if the message byte equals 255 and there are seven bytes between the branch instruction and RTS.



Next, in the fifth line, the program uses indexed addressing to store the byte in the accumulator (the message byte) at the address given in the instruction plus X.

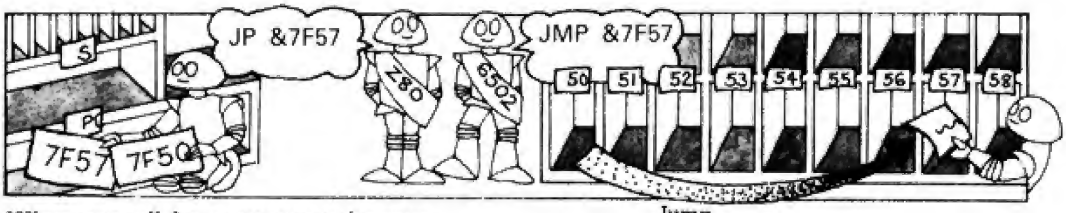
INX stands for "increment X" and it makes the computer add 1 to the contents of the X register. Then it jumps back to the second instruction. This time X is 1, so it loads the next byte of the message into the accumulator and stores it at the next screen location.

Jumping and branching

Making the computer go to an instruction in another part of the program is called branching. There are three different ways of branching: jumps, subroutines and conditional branches. In a conditional branch the computer carries out a test and then branches, or goes on with the next instruction, depending on the result of the test. You can find out more about conditional branches over the page. Jumps just tell the computer to go to a certain address.

The program counter

The program counter is a special 16-bit register which holds the address of the next instruction the computer is to carry out. The computer reads the number in the program counter and then goes to the location with that address to fetch its next instruction. Then the program counter is increased by one so it points to the next memory location.

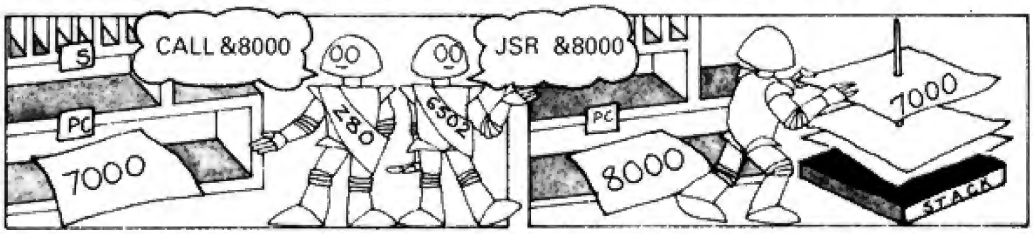


Jump

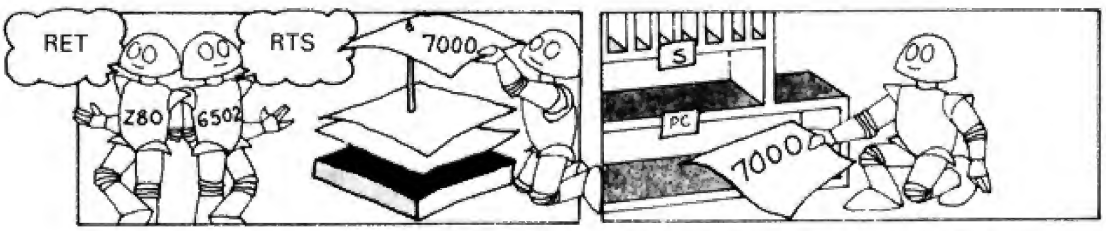
When you tell the computer to jump or branch to a certain address, that address is put in the program counter and the computer then carries out the instructions in sequence from that address. The opcodes for a jump on the Z80 and 6502 are shown in the picture above.

Subroutines

The instruction "CALL address" on the Z80 and "JSR address" (jump to subroutine) on the 6502, tell the computer to go to a subroutine. This is just like in BASIC and at the end of the subroutine you need the return instruction (RET on the Z80 and RTS on the 6502).



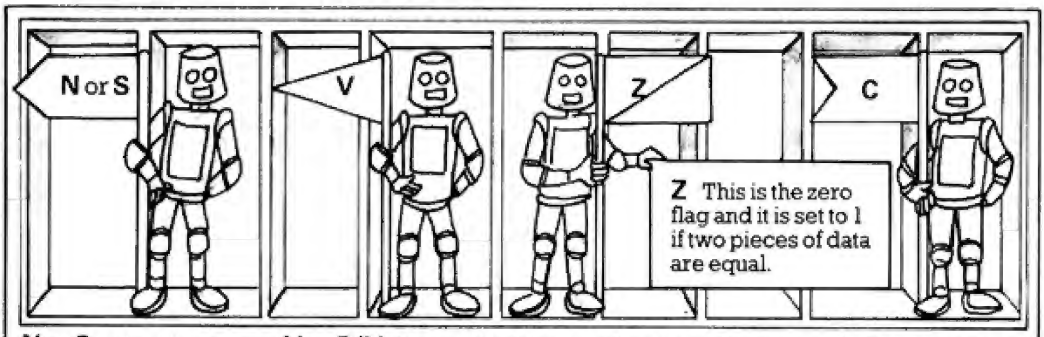
When you tell the computer to go to a subroutine, the address of the subroutine is put in the program counter. The contents of the program counter (the address of the instruction after CALL or JSR) are stored or "pushed" on the stack. The stack is a special part of RAM set aside for the computer's use (see page 10).



When the computer reaches the RTS or RET instruction at the end of the subroutine, it retrieves, or "pops", the last item off the stack and puts it in the program counter. This is the address of the instruction after the one which sent it to the subroutine. This is also what happens when you tell the computer to run a machine code program.

Conditional branches

In a conditional branch the computer tests one of the bits in the flags register and then, depending on the result, either branches or carries on with the next instruction. Here are the bits in the flag register which you can test in conditional branches.



N or S This is the sign bit. It is referred to as N on the 6502 and S on the Z80. It is set to 1 when the result of a calculation is negative and 0 for positive results.

V or P/V This is called the overflow bit on the 6502. On the Z80 it has two functions and is called the parity/overflow. As an overflow bit it is set to 1 when the result of a calculation in two's complement notation (see opposite) results in a carry over to the sign bit.

As a parity bit it is set to 1 if there is an odd number of ones in a byte and is used for checking purposes.

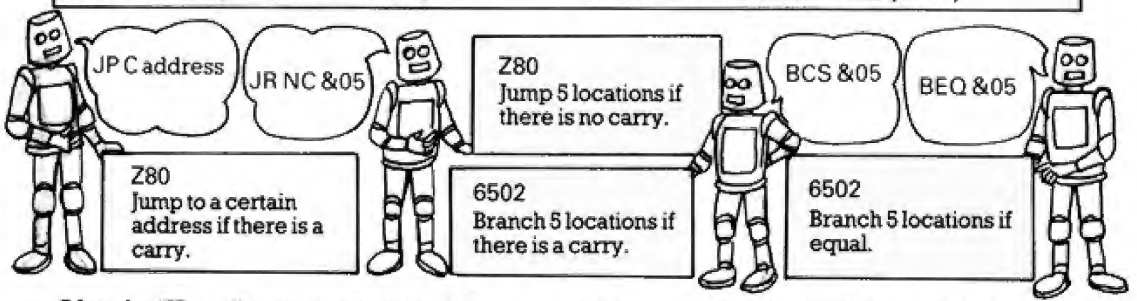
C This is the carry flag. It is set to 1 when the answer to a sum will not fit in one byte.

Various instructions in addition to the compare instruction cause these flags to be automatically set or cleared. For example, on the 6502 the instruction DEC (decrement) affects the sign and zero flags.*

Conditional branch opcodes

Here are the conditional branch instructions for testing each bit.

Z80	6502
Jump if . . .	**Branch if . . .**
JP C .....there is a carry (C = 1).	BCS .....there is a carry (C = 1).
JP NC .....no carry (C = 0)	BCC .....no carry (C = 0)
JP Z .....equal (Z = 1)	BEQ .....equal (Z = 1)
JP NZ.....not equal (Z = 0)	BNE .....not equal (Z = 0)
JP M .....minus (S = 1)	BMI.....minus (N = 1)
JP P .....plus (S = 0)	BPL.....plus (N = 0)
JP PO .....parity odd (P/V = 1)	BVS .....overflow set (V = 1)
JP PE .....parity even (P/V = 0)	BVC .....overflow clear (V = 0)



After the "JP test" instruction on the Z80 you give the computer the address of the instruction you want it to jump to. On the 6502 you give the computer a number which tells it how many locations it has to jump forwards or backwards to find the instruction. This is called "relative addressing" and the number is called the "displacement", or "offset".

The Z80 has an additional conditional branch instruction, "JR test", which you use with a displacement rather than an address. JR stands for "jump relative" and you can only test the zero flag and the carry flag with JR.

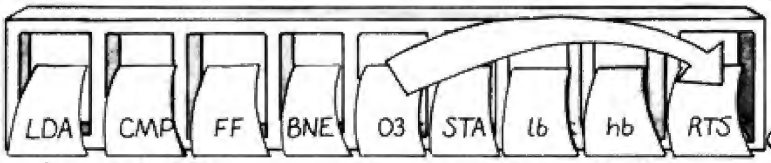
*A complete list of your microprocessor's instruction set will tell you which instructions affect which flags.

Working out the displacement

Remember to count two bytes for an address.

When you give the computer a displacement number in a conditional branch, the computer works out the address of the instruction it is to jump to by adding or subtracting the displacement from the program counter. To work out the displacement, count the number of bytes up to and including the instruction you want to jump to. Start at the instruction after the conditional branch and count that as 0 (because the program counter will already point to that instruction). For example, here are two short 6502 programs which show how you work out the displacement. (The method is the same for the Z80.)

```
LDA address
CMP #&FF
BNE to RTS
STA address
RTS
```

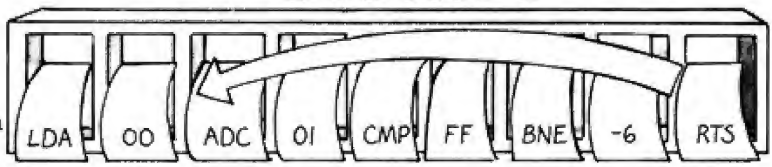


To make the computer jump to the RTS instruction in the example above, the displacement is 3.

In the example below, the displacement to make the computer jump back to the ADC instruction is −6.

```
LDA #&00
ADC #&01
CMP #&FF
BNE to ADC
RTS
```

Count this instruction as 0.



Forwards and backwards jumps

For forwards jumps you just translate the displacement into a hex number and insert it in the program. For backwards jumps, though, the displacement is a negative number and there is no way of indicating negative numbers in eight bit binary. Instead, you use a different system of notation called "two's complement". In two's complement, the left-hand bit is used as a sign bit. If this bit is 1 the number is negative. If it is 0 it is a positive number.

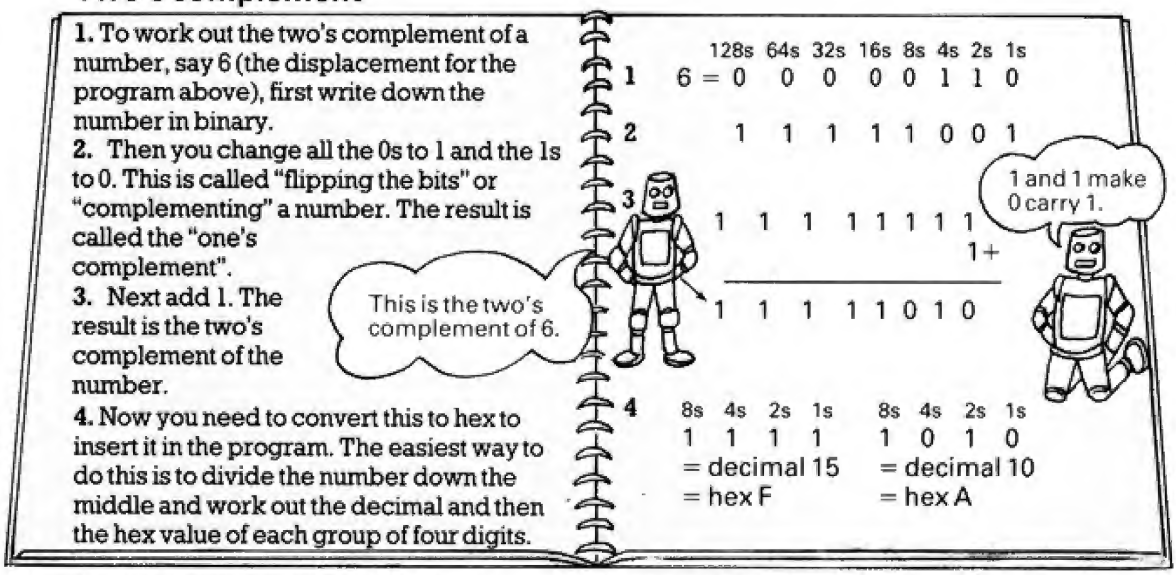
Two's complement

1. To work out the two's complement of a number, say 6 (the displacement for the program above), first write down the number in binary.

2. Then you change all the 0s to 1 and the 1s to 0. This is called "flipping the bits" or "complementing" a number. The result is called the "one's complement".

3. Next add 1. The result is the two's complement of the number.

4. Now you need to convert this to hex to insert it in the program. The easiest way to do this is to divide the number down the middle and work out the decimal and then the hex value of each group of four digits.



So the hex representation of the two's complement of 6 is FA and for a backwards jump you insert this number in the program. In two's complement, the highest number you can represent is 128. This is the biggest backwards displacement you can have. The biggest forwards displacement is 127, the highest number you can make with the eighth binary digit set to 0 to indicate a positive number.

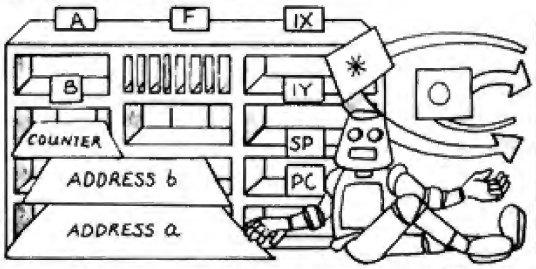
Can you work out the hex for the two's complement of 12, 18 and 9? (Answer page 48)

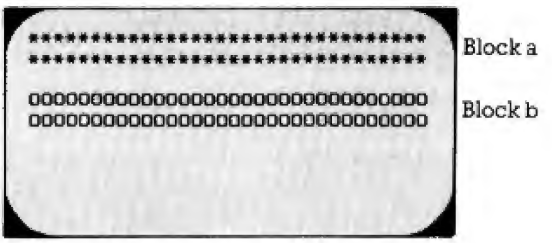
Screen flash program

On these two pages there is a program which swaps two blocks of display on the screen to make a flashing effect. It shows how simple animation works. The program for the Z80 is given below and the one for the 6502 is on the opposite page. At the end there are guidelines for running the program for both microprocessors.

Z80 screen flash

Put very simply, the program swaps the two blocks of the diplay by loading a byte from each block into the registers, then storing the byte from block b in the screen address for block a and vice versa.



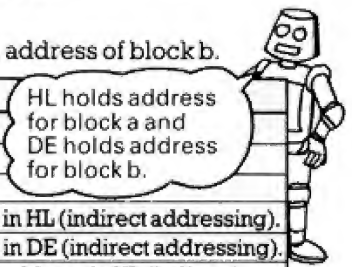


The program uses indirect addressing. The screen addresses for the first byte of each block are stored in registers HL and DE. The computer reads the addresses in these registers each time it loads or stores the bytes. After swapping two bytes the instruction INC (mnemonic for increment) makes it add one to HL and DE so that when the program repeats, these are the addresses of the next two bytes in each block on the screen.

Register B holds the number of bytes to be swapped. Each time the program repeats, B is decremented (decreased) by 1 so it acts as a counter. When B=0 all the bytes have been swapped.

Z80 program

n=number of bytes in one block; a=first address of block a; b=first address of block b.



Mnemonics	Hex codes	Meaning
LD B, n	06, n	Counter.
LD HL, (address a)	21, address a	Put address of block a in HL.
LD DE, (address b)	11, address b	Put address of block b in DE.
LD C, (HL)	4E	Load C with contents of address in HL (indirect addressing).
LD A, (DE)	1A	Load A with contents of address in DE (indirect addressing).
LD (HL), A	77	Store contents of accumulator at address in HL (indirect).
LD A, C	79	Put C (first byte block a) into accumulator.
LD (DE), A	12	Store contents of accumulator at address in DE.
INC HL INC DE	23 13	Add one to HL and DE.
DEC B	05	Decrement B, the counter.
LD A, &00	3E, 00	Put 0 in the accumulator
CP B	B8	Compare B with contents of the accumulator (0).
JR NZ to 4th instruction	20, F3	If B does not equal zero, jump back &F3 locations to load next bytes into registers. F3 is hex for two's complement of 13 (see page 37).
RET	C9	Return.

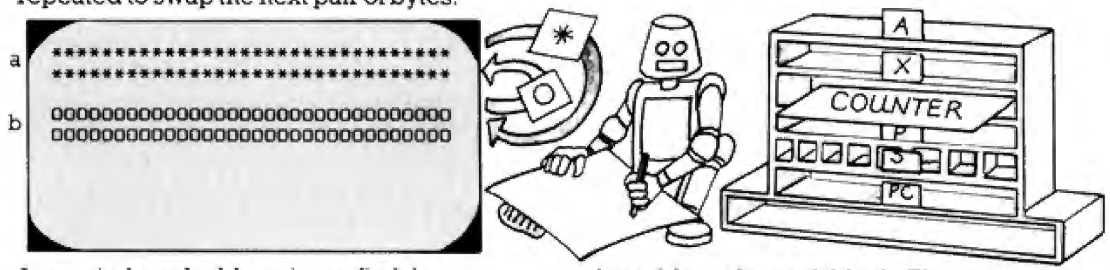
Filling in the data and addresses

n (number of characters in one block) To find n, multiply the number of characters in a line by the number of lines in one block. Convert to hex.

addresses a and b If you want to swap the top two lines of the screen with the next two lines, make address a the first address of your computer's screen memory. Address b is the address for block a plus the number of bytes to be swapped. Convert both addresses to hex.

6502 screen flash

This program swaps the two blocks, byte by byte (i.e. character by character), starting with the last byte in each block. It loads these bytes into the registers, then stores the byte from block a in the screen location for block b and vice versa. Then the program is repeated to swap the next pair of bytes.



It uses indexed addressing to find the address for each byte. The total number of bytes in one block is loaded into the X register. Then, to store or load a byte, the number in the X register is added to the starting address for each block. The instruction DEX (decrement X) makes the computer subtract 1 from X so that, when the program repeats, the computer fetches the next byte back in the display.

6502 screen flash program

See the bottom of the opposite page for how to work out the values of n, a and b. Then subtract 1 from a and b so that when the computer adds X it gets the last address in each block, rather than the first address of the next line. (Make sure n, a and b are in hex.)

Mnemonics	Hex codes	Meaning
LDX #n	A2 n	Load X with the number of bytes in one block.
LDA address a, X	BD address a	Put contents of location with address a + X into accumulator.
TAY	A8	Transfer contents of accumulator to register Y.
LDA address b, X	BD address b	Put contents of location with address b + X into accumulator.
STA address a, X	9D address a	Store contents of accumulator at address a + X.
TYA	98	Transfer contents of Y register back to accumulator.
STA address b, X	9D address b	Store contents of accumulator at address b + X.
DEX	CA	Decrement X. Zero flag is set to 1 when X = 0.
BNE to instruction two	D0 EF	Branch back &EF locations if X is not equal to 0. EF is the hex for two's complement of 17 (see page 37).
RTS	60	Return

Loading and running the program for the Z80 or 6502

The best way to run this program is as a machine code subroutine in the hex loader. To do this, follow these steps:

1. Type in the hex loader and put the hex codes for your computer's microprocessor in line 160.

2. At line 180 you need two loops to poke the characters for the display into the screen memory. For example, here are the lines for two rows of *s (code 42) followed by two rows of 0s (code 48), for a computer with a 40 column screen.

```
180 FOR J=0 TO 79
190 POKE first screen address + J,42
200 NEXT J
210 FOR J=80 TO 159
220 POKE first screen address + J,48
230 NEXT J
```

3. Next, add the following lines to the end of the program:

```
240 CALL address where machine
code is stored
250 FOR K=1 TO 500
260 NEXT K
270 GOTO 240
```

Change figure 500 in delay loop to suit your computer.

4. Now type RUN to run the program. The hex loader pokes the hex codes into the memory, then pokes the display codes into the screen memory. Line 240 makes it go to the location where the machine code program is stored and carry out the instructions. By itself, the machine code program only swaps the display once, so line 270 makes it call the program again and again to make a flashing effect. You need the delay loop because the machine code is so fast.

Going further

If you want to find out more about machine code the best way is to try writing your own short programs and to test and study programs written by other people. One good way to use machine code is as a short subroutine to carry out a particular task in a BASIC program. For instance, machine code is particularly suitable for sorting data or filling the screen with graphics because it is faster and takes less memory space than BASIC. You can find subroutines for doing things like this in magazines. If the subroutines are written specially for your computer you can run them without alteration. If they are written for another make of computer which uses the same microprocessor you will need to change any addresses in the program for addresses in the area in your computer's memory that you have chosen to store machine code.

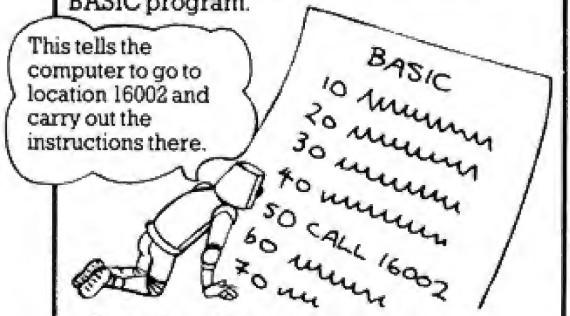
Machine code subroutines

Here are the steps you need to follow to use a machine code subroutine in a BASIC program.

1. Make room in the memory for the machine code by lowering the top of user RAM (see pages 20-22).

2. Put the codes for the machine code subroutine into line 160 of the hex loader program on page 24. (Make sure there is a return instruction at the end of the machine code program.) Add lines to poke in any data bytes if necessary, then type in and run the hex loader.

3. Number your BASIC program using line numbers starting after those used in the hex loader. At the point where you want the computer to carry out the machine code, put your computer's command for running a machine code program as a line in the BASIC program.



4. Type the BASIC program into your computer and then type RUN. The computer will carry out the BASIC instructions and when it reaches the line telling it to run the machine code program it will go to the address where the machine code is stored and carry out the instructions. The return instruction at the end of the machine code will send the computer back to the next line in the BASIC program.

Using an assembler

An assembler (a program which enables you to type in a machine code program in mnemonics) makes machine code programming much easier. You can buy an assembler on cassette for most home computers and some, such as the BBC, have a built-in assembler.

With an assembler you can type in comments alongside the mnemonics to remind you what each line does. The assembler will then display the program on the screen in hex and mnemonics, with the addresses where the instructions are stored and the comments.

The assembler will automatically reverse the pairs of digits in addresses and work out the address or displacement for a jump. Some assemblers allow you to use symbolic names for data, like variables in BASIC. A good assembler also has a debugger to find mistakes and an editor to help you correct them.

Suggested books

There are lots of books on machine code specially written for one particular make of microcomputer. The best way to choose one is to read the reviews in computer magazines. You may also find the following books useful:

Programming the Z80 and *Programming the 6502*, both by Rodney Zaks and published by Sybex. These are very detailed guides with complete lists of all the instructions for each microprocessor. They are not easy to read for beginners, but they are useful for reference.

VIC 20 Programmer's Reference Guide published by Commodore.

6502 Machine Code for Beginners by A. P. Stephenson, Newnes Microcomputer Books.

Decimal/hex conversion charts

This chart converts hex numbers from 0 to FF to decimal and vice versa.

Hex to decimal

To convert a hex number to decimal read along the row for the first hex digit in your hex number and down the column for the second hex digit. The number where the row and column meet is the decimal equivalent for your hex number, e.g. hex A1 is decimal 161.

Decimal to hex

To convert a decimal number to hex, find the decimal number in the chart. Then read back along the row for the first hex digit and up the column for the second hex digit e.g. 154 is 9A.

First hex digit \ Second hex digit	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
8	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
9	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
A	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
B	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
C	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
D	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
E	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
F	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

Converting addresses

To use the chart to convert hex addresses, look up the decimal equivalent for the first pair of digits in the address. This is the page number. Then look up the decimal equivalent for the second pair of digits to find the position on the page. Multiply the page number by 256 and add the position on the page.

Two's complement conversion chart

This chart gives the two's complement in hex of decimal numbers from −1 to −128. To convert a number to two's complement, find the number in the chart, then read along the row for the first hex digit and up the column for the second digit.

First hex digit \ Second hex digit	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
F	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
E	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
D	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48
C	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64
B	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80
A	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96
9	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111	112
8	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127	128

Doing conversions on a calculator

When you do conversions on a calculator the calculator displays the remainder as a decimal number. For example, if you are converting decimal 134 to hex you divide by 16 then convert the answer and remainder to hex digits. A calculator would give you the answer as 8.375.

To convert the remainder to a whole number you subtract the number before the decimal point, then multiply by the number you divided by.

$8.375 - 8 = 0.375 \times 16 = 6$

So $134 \div 16 = 8$ remainder 6 therefore decimal 134 is 86 in hex.

Z80 mnemonics and hex codes

The mnemonics and hex codes for the instructions covered in this book are given on the next few pages. The term "implicit addressing" used in these lists is just the name for instructions where no operand need be specified in the hex code. There are a few other instructions not listed here and if you want to go further with machine code you will need a complete list of the Z80 instruction set (see page 40). The following abbreviations are used in these lists:

n = number
nn = two byte number
r = register
rr = register pair
x = address
c = condition
d = displacement

ADC A,n Add with carry, a number, n, to the accumulator. (Immediate addressing.)

ADC A,n	CE,n

ADC A,r Add with carry, register r to the accumulator. (Implicit addressing.)

ADC A,A	8F
ADC A,B	88
ADC A,C	89
ADC A,D	8A
ADC A,E	8B
ADC A,H	8C
ADC A,L	8D

ADC HL,rr Add with carry, the contents of register pair rr to HL. (Implicit addressing.)

ADC HL,BC	ED4A
ADC HL,DE	ED5A
ADC HL,HL	ED6A

ADD A,n Add a number, n, to the accumulator. (Immediate addressing.)

ADD,n	C6,n

ADD A,r Add register r to the accumulator. (Implicit addressing.)

ADD A,A	87
ADD A,B	80
ADD A,C	81
ADD A,D	82
ADD A,E	83
ADD A,H	84
ADD A,L	85

ADD HL,rr Add the contents of register pair rr to HL. (Implicit addressing.)

ADD HL,BC	09
ADD HL,DE	19
ADD HL,HL	29

CALL x Go to subroutine starting at address x. (Immediate addressing.)

CALL x	CD x

CALL c,x Go to subroutine starting at address x depending on condition c. c may be Z (equal); NZ (not equal); C (carry); NC (no carry); PE (parity even); PO (parity odd); M (minus) or P (plus). (Immediate addressing.)

CALL Z,x	CC,x
CALL NZ,x	C4,x
CALL C,x	DC,x
CALL NC,x	D4,x
CALL PE,x	EC,x
CALL PO,x	E4,x
CALL M,x	FC,x
CALL P,x	F4,x

CCF Complement carry flag. (Implicit addressing.)

CCF	3F

CP n Compare contents of accumulator with data n. (Immediate addressing.)

CP n	FE n

CP r Compare contents of register r with the accumulator. (Implicit addressing.)

CP A	BF
CP B	B8
CP C	B9
CP D	BA
CP E	BB
CP H	BC
CP L	BD

CP (HL) Compare contents of accumulator with contents of address held in HL. (Indirect addressing.)

CP (HL)	BE

DEC r Decrement register r. (Implicit addressing.)

DEC A	3D
DEC B	05
DEC C	0D
DEC D	15
DEC E	1D
DEC H	25
DEC L	2D

DEC rr Decrement register pair rr. (Implicit addressing.)

DEC BC	0B
DEC DE	1B
DEC HL	2B
DEC IX	DD2B
DEC IY	FD2B

DEC (HL) Decrement contents of address held in HL. (Indirect addressing.)

DEC (HL)	35

INC r Increment register r. (Implicit addressing.)

INC A	3C
INC B	04
INC C	0C
INC D	14
INC E	1C
INC H	24
INC L	2C

INC rr Increment register pair rr. (Implicit addressing.)

INC BC	03
INC DE	13
INC HL	23

INC (HL) Increment contents of address held in HL. (Indirect addressing.)

INC (HL)	34

JP x Jump to address x. (Immediate addressing.)

JP x	C3 x

JP (rr) Jump to address held in register pair rr. (Implicit addressing.)

JP (HL)	E9
JP (IX)	DDE9
JP (IY)	FDE9

JP c,x Jump to address x depending on condition c. c may be Z (equal); NZ (not equal); C (carry); NC (no carry); PE (parity even); PO (parity odd); M (minus) or P (plus). (Immediate addressing.)

JP Z,x	CA,x
JP NZ,x	C2,x
JP C,x	DA,x
JP NC,x	D2,x
JP PE,x	EA,x
JP PO,x	E2,x
JP M,x	FA,x
JP P,x	F2,x

JR d Jump relative. Jump d bytes (the displacement). (Relative addressing.)

JR d	18 d

JR c,d Jump relative. Jump d bytes (the displacement) depending on condition c. c may be NZ (not equal); Z (equal); NC (no carry) or C (carry). (Relative addressing.)

JR NZ,d	20,d
JR Z,d	28,d
JR NC,d	30,d
JR C,d	38,d

LD r,n Load register r with data n. (Immediate addressing.)

LD A,n	3E,n
LD B,n	06,n
LD C,n	0E,n
LD D,n	16,n
LD E,n	1E,n
LD H,n	26,n
LD L,n	2E,n

LD rr,nn Load register pair rr with two byte number nn. (Immediate addressing.)

LD BC,nn	01,nn
LD DE,nn	11,nn
LD HL,nn	21,nn

LD A,(x) Load accumulator with contents of address x. (Absolute addressing.)

LD A,(x)	3A,(x)

LD rr,(x) Load register pair rr with contents of addresses x and x+1. (Absolute addressing.)

LD BC,(x)	ED4B,(x)
LD DE,(x)	ED5B,(x)
LD HL,(x)	2A,(x)

LD A,r Load the accumulator with contents of register r. (Implicit addressing.)

LD A,A	7F
LD A,B	78
LD A,C	79
LD A,D	7A
LD A,E	7B
LD A,H	7C
LD A,L	7D

LD B,r Load register B with the contents of register r. (Implicit addressing.)

LD B,A	47
LD B,B	40
LD B,C	41
LD B,D	42
LD B,E	43
LD B,H	44
LD B,L	45

LD C,r Load register C with the contents of register r. (Implicit addressing.)

LD C,A	4F
LD C,B	48
LD C,C	49
LD C,D	4A
LD C,E	4B
LD C,H	4C
LD C,L	4D

LD D,r Load register D with the contents of register r. (Implicit addressing.)

LD D,A	57
LD D,B	50
LD D,C	51
LD D,D	52
LD D,E	53
LD D,H	54
LD D,L	55

LD E,r Load register E with the contents of register r. (Implicit addressing.)

LD E,A	5F
LD E,B	58
LD E,C	59
LD E,D	5A
LD E,E	5B
LD E,H	5C
LD E,L	5D

LD H,r Load register H with the contents of register r. (Implicit addressing.)

LD H,A	67
LD H,B	60
LD H,C	61
LD H,D	62
LD H,E	63
LD H,H	64
LD H,L	65

LD L,r Load register L with the contents of register r. (Implicit addressing.)

LD L,A	6F
LD L,B	68
LD L,C	69
LD L,D	6A
LD L,E	6B
LD L,H	6C
LD L,L	6D

LD r,(rr) Load register r with contents of address held in register pair rr. (Indirect addressing.)

LD A,(BC)	0A
LD A,(DE)	1A
LD A,(HL)	7E
LD B,(HL)	46
LD C,(HL)	4E
LD D,(HL)	56
LD E,(HL)	5E
LD H,(HL)	66
LD L,(HL)	6E

LD (x),A Store the contents of the accumulator in address x. (Absolute addressing.)

LD (x),A	32,x

LD (x),rr Store the contents of register pair rr at addresses x and x+1. (Absolute addressing.)

LD (x),BC	ED43,x
LD (x),DE	ED53,x
LD (x),HL	22,x

LD (rr),r Store the contents of register r at the address held in register pair rr. (Indirect addressing.)

LD (BC),A	02
LD (DE),A	12
LD (HL),A	77
LD (HL),B	70
LD (HL),C	71
LD (HL),D	72
LD (HL),E	73
LD (HL),H	74
LD (HL),L	75

LD (rr), n Store data n at address held in register pair rr. (Immediate/indirect addressing.)

LD (HL),n	36

RET Return from subroutine. (Indirect addressing.)

RET	C9

RET c Return from subroutine depending on condition c. c can be Z (equal); NZ (not equal); C (carry); NC (no carry); PE (parity even); PO (parity odd); P (plus); M (minus). (Indirect addressing.)

RET Z	C8
RET NZ	C0
RET C	D8
RET NC	D0
RET PE	E8
RET PO	E0
RET M	F8
RET P	F0

SBC A,n Subtract with carry data n from the accumulator. (Immediate addressing.)

SBC A,n	DE,n

SBC A,r Subtract with carry contents of register r from the accumulator. (Implicit addressing.)

SBC A,A	9F
SBC A,B	98
SBC A,C	99
SBC A,D	9A
SBC A,E	9B
SBC A,H	9C
SBC A,L	9D

SBC HL,rr Subtract with carry contents of register pair rr from register pair HL. (Implicit addressing.)

SBC HL,BC	ED42
SBC HL,DE	ED52
SBC HL,HL	ED62

SBC A,(HL) Subtract with carry the contents of address held in register pair HL, from the accumulator. (Indirect addressing).

SBC A,(HL)	9E

SCF Set carry flag. (Implicit addressing.)

SCF	37

SUB n Subtract data n from the accumulator. (Immediate addressing.)

SUB, n	D6, n

SUB r Subtract contents of register r from the accumulator. (Immediate addressing.)

SUB A	97
SUB B	90
SUB C	91
SUB D	92
SUB E	93
SUB H	94
SUB L	95

SUB (HL) Subtract the contents of address held in HL from the accumulator. (Indirect addressing.)

SUB (HL)	96

Puzzle answers

Page 11

&A7 in decimal is 167. 513 in hex is &201.

Page 26

1. 25+73 (25 is &19 and 73 is &49)

Z80		6502		Meaning
Mnemonics	**Hex codes**	**Mnemonics**	**Hex codes**	
LD A, &19	3E,19	LDA #&19	A9 19	Put &19 in accumulator.
ADD A, &49	C6,49	ADC #&49	69 49	Add &49 to accumulator.
LD (address), A	32, address	STA address	8D address	Store contents of accumulator at a certain address.
RET	C9	RTS	60	Return

2. 64+12+14 (64 is &40, 12 is &0C and 14 is &0E)

Z80		6502		Meaning
Mnemonics	**Hex codes**	**Mnemonics**	**Hex codes**	
LD A, &40	3E,40	LDA #&40	A9 40	Put &40 in accumulator.
ADD A, &0C	C6,0C	ADC #&0C	69 0C	Add &0C to accumulator.
ADD A, &0E	C6,0E	ADC #&0E	69 0E	Add &0E to accumulator.
LD(address), A	32, address	STA address	8D address	Store contents of accumulator at a certain address.
RET	C9	RTS	60	Return

Tip: an easy way to work out the two's complement of a number is to subtract it from 256, then convert the answer to hex. E.g. 256 – 6 = 250 which is FA in hex.

Puzzle answers continued on page 48.

6502 mnemonics and hex codes

This chart shows the mnemonics and hex codes for all the instructions (plus a few more) covered in this book. The mnemonic instructions are given down the left and the hex codes for each instruction in the different addressing modes are shown across the chart. Zero page addressing is just like absolute addressing, i.e. the operand is the address where the data is stored, but the address must be in page zero (i.e. locations 0-255) of the memory (see page 10). Implied addressing is just the term used to describe instructions where no operand need be specified, e.g. CLC. There are a number of other instructions not given here, and if you want to go further with machine code you will need to get a complete list of the 6502 instruction set.

Addressing mode	*Immediate*	*Absolute*	*Zero Page*	*Indexed X*	*Indexed Y*	*Implied*	*Relative*
Operand is	Data	Any address	Address in page zero	Address + X register	Address + Y register	None	Displacement
ADC Add with carry, i.e. add a byte, plus the carry flag, to the accumulator.	69	6D	65	7D	79		
BCC Branch if carry clear.							90
BCS Branch if carry set.							B0
BEQ Branch if equal.							F0
BMI Branch if minus.							30
BNE Branch if not equal.							D0
BPL Branch if plus.							10
BVC Branch if overflow clear.							50
BVS Branch if overflow set.							70
CLC Clear carry flag.						18	
CMP Compare with the accumulator.	C9	CD	C5	DD	D9		
CPX Compare with register X.	E0	EC	E4				
CPY Compare with register Y.	C0	CC	C4				
DEC Decrement (subtract 1 from) memory location.		CE	C6	DE			
DEX Decrement (subtract 1 from) X register.						CA	
DEY Decrement (subtract 1 from) Y register.						88	
INC Increment (add 1 to) memory location.		EE	E6	FE			
INX Increment (add 1 to) X register.						E8	
INY Increment (add 1 to) Y register.						C8	
JMP Jump to address specified in operand.		4C					
JSR Jump to subroutine starting at address specified in operand.		20					
LDA Load accumulator.	A9	AD	A5	BD	B9		
LDX Load X register.	A2	AE	A6		BE		
LDY Load Y register.	A0	AC	A4	BC			
RTS Return from subroutine.						60	
SBC Subtract with carry. Subtract from the accumulator and borrow from the carry flag.	E9	ED	E5	FD	F9		
SEC Set carry flag.						38	
STA Store accumulator at a certain address.		8D	85	9D	99		
STX Store X register at a certain address.		8E	86				
STY Store Y register at a certain address.		8C	84				
TAX Transfer accumulator to X register.						AA	
TAY Transfer accumulator to Y register.						A8	
TXA Transfer X register to accumulator.						8A	
TYA Transfer Y register to accumulator.						98	

Note that not all the instructions can be used in all the addressing modes.

Machine code words

 # Hash sign. This is the sign used on some computers to indicate hex numbers. For the 6502 microprocessor it is used to indicate a piece of data.
& Ampersand sign. This is another sign used to indicate hex numbers.
Absolute address. The actual address of a piece of data.
Absolute addressing. An addressing mode in which the instruction contains the address of the data. Also called extended or direct addressing.
Accumulator. The register where bytes of information on which arithmetical or logical operations are to be carried out, are held.
Address. A number used to identify a location in the computer's memory.
Addressing modes. The various ways in which you can tell the computer where to find the data to work on in a machine code program.
Arithmetic logic unit (ALU). The area inside the CPU where arithmetical and logical operations are carried out.
Assembler. A program which converts instructions written in assembly language mnemonics into the computer's own code.
Assembly language. A method of programming the computer using letter codes, called mnemonics, to represent machine code instructions.
Binary. A number system with two digits, 0 and 1 and in which each digit in a number has twice the value of the digit on its right.
Bit. A single unit of computer code, i.e. a 1 or 0 representing a pulse or no-pulse signal.
Buffer. A temporary storage area in the computer's memory where data is held on its way to or from its final destination.
Branch. An instruction telling the computer to jump to another line in a program.
Byte. A group of eight pulse and no-pulse signals (or "bits") which represents a piece of information in computer code.
Carry flag. A bit in the flags register which is set to 1 when the result of an addition will not fit into eight bits.
Clear. To make a bit, e.g. one of the bits in the flags register, zero.
Complement. Also called "flipping the bits" this is the process of changing all the 0s in a byte to 1 and all the 1s to 0.
Conditional branch. An instruction which tells the computer to jump to another line in the program depending on the result of a test.
Direct addressing. See absolute addressing.
Disassembler. A program which can display the contents of a series of memory locations on the screen in assembly language. You can buy a disassembler on cassette and it is useful for debugging machine code programs and for examining the programs in your computer's ROM.
Displacement. A number used in a jump or branch instruction to tell the computer how many locations to jump to find the next instruction. Also called an offset.
Flag. A bit in the flags register which is used to indicate a certain condition, e.g. the presence of a negative number, or of a carry over in an addition.
Hexadecimal, or hex. A number system which uses 16 digits (the numbers 0-9 and letters A-F). Each digit in a hex number has 16 times the value of the digit on its right.
Hex loader. A BASIC program which converts the hex codes of a machine code program into decimal numbers and pokes them into the computer's memory.
High order byte. The first two digits in a hex address which represent the number of the page in the memory where the address is. Also, the two digits which show how many 256s there are in a number larger than 255.
HIMEM. The highest address in user RAM.
Immediate addressing. An addressing mode in which the data for an instruction is included in the instruction.
Implicit addressing. An addressing mode in which the operand is understood and need not be specified.
Implied addressing. Same as implicit, see above.
Indexed addressing. An addressing mode in which the contents of an index register are added to the address given in the instruction to work out the actual address of the data.
Index registers. The registers used in indexed addressing and also, in the 6502, as general purpose registers.
Indirect addressing. An addressing mode in which the operand is used as a

pointer to the data. The operand may be an address or, in the Z80, a pair or registers, and it holds the address of the data.

Instruction. An operation to be carried out by the central processing unit.

Interpreter. A program which translates instructions in BASIC (or other high level language) into the computer's own code.

Instruction set. All the operations which can be carried out by a particular microprocessor.

Jump. An instruction which tells the computer to go to another line in the program.

LIFO. This stands for "last in/first out" and describes the method used by the computer to store information in the stack.

Low order byte. The two hex digits in an address which give the position of that address within a page of memory. Also, the two hex digits which show the number of units in a number larger than 255.

Microprocessor. The chip which contains the computer's CPU and which carries out program instructions and controls all the other activities inside the computer.

Mnemonic. A letter code used in assembly language to represent an instruction in the computer's own code. The word mnemonic (pronounced nemonic) means "to aid the memory" and assembly language mnemonics sound like the instructions they represent.

Object code. A program which has been translated into machine code from assembly language or another high level language.

Offset. See displacement.

Opcode. The part of an instruction which tells a computer what to do.

Operand. The part of an instruction which tells the computer where to find the data to work on.

Operating system. A group of programs written in machine code and stored in the computer's ROM, which tell it how to carry out all the tasks it has to do.

Page. A subdivision of memory. On most home computers a page is 256 locations.

Pointer. A memory location (or pair of registers) which contains the address of a piece of data.

Pop. To remove an item stored in the stack.

Processor status register. This is the 6502 name for the flags register (the register where each bit is used to record a certain condition inside the computer).

Program counter. The register which contains the address of the next instruction to be fetched from the memory.

Pull. Same as pop, i.e. to remove an item from the stack.

Push. To place an item in the stack.

RAMTOP. The highest address in user RAM.

Registers. The places in the CPU where bytes of instructions, data and addresses are held while the computer works on them.

Relative addressing. An addressing mode in which the computer works out the address of the next instruction by adding a number called the displacement or offset, to the address in the program counter.

Screen memory. The locations in RAM which are used to hold information to be displayed on the screen.

Sign flag. The bit in the flags register which is used to indicate negative and positive numbers.

Source code. A program written in assembly language, or other high level language such as BASIC.

Stack. An area of the memory used by the computer for temporary storage and where the last item stored is always the first to be retrieved.

Stack pointer. A register in the CPU which contains the address of the last item in the stack.

Systems variables. Memory locations in RAM which hold information about the current state of the computer.

Top of memory. The highest address in user RAM.

Two's complement. A system of notation used to represent negative numbers. To find the two's complement of a number you complement (make all the 1s into 0s and all the 0s into 1s) the binary for that number and then add 1.

User RAM. The part of RAM where BASIC programs are stored.

Zero flag. The bit in the flags register which indicates when the result of an operation is 0 and is also used to show when two bytes are equal.

Zero page. The first 256 locations in the memory.

Zero page addressing. Used only on the 6502, this is an addressing mode in which the operand is an address in page zero of the memory (i.e. from 0-255).

Puzzle answers continued

Page 28

00011010 is 26 decimal.
11111011 is 251 decimal.
10101010 is 170 decimal.

	Decimal		Hex	
	High order	Low order	High order	Low order
307	1	51	&01	&33
21214	82	222	&82	&DE
759	2	247	&02	&F7
1023	3	255	&03	&FF

Page 31

To adapt the program on page 27 for answers greater than 255 you need to delete the return instruction and add the lines given below. To see the result you use this command:
PRINT PEEK(address 3) + PEEK(address4)*256.

Z80		6502		Meaning
Mnemonics	**Hex codes**	**Mnemonics**	**Hex codes**	
LD A, &00	3E,00	LDA #&00	A9 00	Put 0 in accumulator.
ADC A, &00	CE,00	ADC #&00	69 00	Add with carry, 0 to accumulator.
LD(address 4),A	32, address 4	STA address 4	8D address 4	Store contents of accumulator at address 4.
RET	C9	RTS	60	Return.

Page 37

Hex for the two's complement of 12 is &F4; 18 is &EE and 9 is &F7.

Index

& ampersand sign, 8, 12, 16, 18, 46
# hash sign, 12, 16, 18, 46
absolute addressing, 18, 27, 46
accumulator, 14-15, 17, 30, 32, 46
address, 8-9, 11, 19, 46
  converting to hex or decimal, 11
  in machine code, 18-19
addressing modes, 27, 46
ALU (arithmetic/logic unit), 13, 14, 46
ASCII code, 13, 24, 32
assembler, 5, 16, 40, 46
assembly language, 5, 17, 19, 46
Atari, 3, 24
BASIC, 4, 12,20, 40
big numbers, 28, 30-32
binary,
  code, 4, 5, 16
  numbers, 4, 19, 28, 46
  to hex conversion, 37
bit, 4, 46
branch, 34, 35, 46
buffers, 10, 46
byte, 4, 13, 19, 20, 46
carry flag, 14, 15, 17, 29, 30, 31, 36, 46
carrying over numbers in addition, 29, 30, 31
character codes, 13, 32
clear, to, 29, 46
Commodore 64, 3, 7
comparing, 32
complement, 46
conditional branches, 35, 36-37, 46
control unit, 13, 14
CPU (central processing unit), 7, 14-15, 16, 19
crash, 20
databytes, 23, 28
decimal numbers, 11, 41
decrement, 36, 38
direct addressing, 27, 46
disassembler, 46
displacement, 36-37, 46
display file, 8
extended addressing, 27
flags register, 14-15, 17, 29, 36
hex,
codes, 16, 18, 19
converting to decimal, 11, 41
dump, 19
loader, 5, 23, 24, 25, 46
number system, 5, 8, 11, 46
high order byte, 19, 28, 30, 31, 46
HIMEM, 8, 20, 21, 46
immediate addressing, 18, 27, 33, 46
implicit addressing, 46
implied addressing, 46
increment, 33, 34, 38
indexed addressing, 34, 39, 46
index registers, 14-15, 46
indirect addressing (Z80), 33, 38, 46
instruction, 4, 5, 13-14, 16, 47
instruction set, 16, 47
interpreter, 4, 8, 20, 47
jumps, 33, 35, 47
LIFO, 10, 47
locations, memory, 8-9, 10, 11, 12-13
lowering RAMTOP, 21
low order byte, 19, 28, 30-31, 47
machine code,
  checklist, 26
  length of program, 20
  subroutines, 39, 40
  where to store in memory, 20-22
memory, 8-9, 10, 12-13
memory map, 8
microprocessor, 7, 16, 47
mnemonics, 5, 16-17, 47
object code, 18, 47
offset, 36-37, 47
opcode, 16, 18, 19, 47
operand, 16, 18, 27, 47
operating system, 8, 10, 11, 13, 20, 47
Oric micro, 3, 7, 21
overflow bit, 36
page (of memory), 10, 11, 19, 21, 47
parity/overflow bit, 36
PEEK, 12-13, 21, 26, 31
pointer, 33, 47
POKE, 12-13, 23
pop, 35, 47
position on page (of address), 11, 19, 21
processor status register, 15, 29, 47
  (see also flags register)
program counter, 14-15, 35, 47
RAM (random access memory), 6, 12, 13, 20-21
RAMTOP, 8, 20, 21, 47
  lowering, 20-22
registers, 13-14, 27, 30, 31, 47
relative addressing, 36, 47
REM statement, storing machine code in, 22
reserved for use of the operating system, 8, 10
return instruction, 23, 35
ROM (read only memory), 6, 12, 13
running a machine code program, 25
screen memory, 8, 13, 47
set, to, 29
sign flag, 14, 36, 47
source code, 18, 47
Spectrum, 13, 24, 32
stack, 10, 14, 15, 20, 35, 47
stack pointer, 14-15, 47
subroutines, 35
systems variables, 10, 20, 21, 47
Timex 1000, 9, 13, 22, 24
Timex 2000, 13, 24, 32
top of memory, 20, 21, 47
two's complement, 37, 41, 47
user RAM, 8, 20, 47
VIC 20, 7, 13, 22
zero flag, 32, 33, 34, 36, 47
zero page, 10, 45, 47
zero page addressing, 45, 47
ZX81, 9, 13, 22, 24

Hex loader conversions

Change these lines for the ZX81 (Timex 1000):

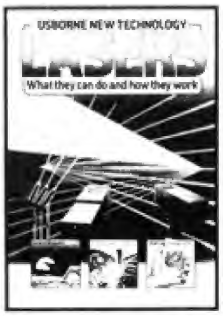
```
 40 INPUT H$
 70 LET X=
    (CODE(H$)-28)*16
 80 Delete
 90 LET Y=CODE
    (H$(2 TO ))-28
100 LET X=X+Y
110 Delete
155 Delete
160 Delete
```

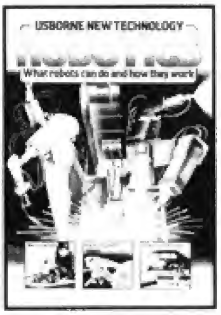
Change this line for Atari computers:

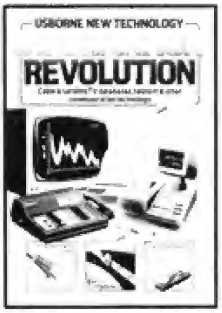
```
90 LET Y=ASC(A$(2))
```

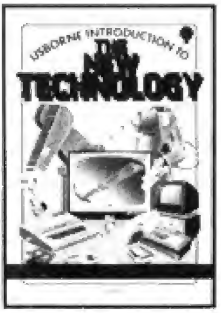
Other Usborne Books

There are hundreds of colourful Usborne books for all ages on a wide range of subjects. Titles which may be of particular interest to you are:









This exciting new series takes a serious look at what is happening *now* in the world of new technology. Many people think that such things as lasers, robots, databases and interactive TV belong only to the world of science fiction but, as these brilliantly illustrated books show, many of them are already in use and affecting our everyday lives. The books take a straightforward approach to these apparently difficult subjects, making them easy for everyone to understand.

Page size: 240 × 170 mm 48 pages

This up-to-the-minute series on electronic technology explores the worlds of computers, TV and video, audio and radio and, in a new title, films and special effects. In a clear visual way, the books describe the very latest equipment and show what it does and how it works. They also explain much of the confusing technical jargon which usually surrounds these subjects. There are fascinating sections on what computers can do for us and how they do it, how TV and video cameras can turn an ordinary scene into a pattern of electronic signals that can be stored on tape, and how a recording studio works. *Audio & Radio* also contains instructions for building a simple radio.

Page size: 276 × 216 mm 32 pages

Usborne Computer Books

Usborne Computer Books are colourful, straightforward and easy-to-understand guides to the world of home computing for beginners of all ages.

Usborne Guide to Computers A colourful introduction to the world of computers. *"Without question the best general introduction to computing I have ever seen."* Personal Computer World

Understanding the Micro A beginner's guide to microcomputers, how to use them and how they work. *"This introduction to the subject seems to get everything right."* Guardian

Computer Programming A simple introduction to BASIC for absolute beginners. *"... lucid and entertaining ..."* Guardian

Computer and Video Games All about electronic games and how they work, with expert's tips on how to win. *"The ideal book to convert the arcade games freak to real computing."* Computing Today

Computer Spacegames, Computer Battlegames Listings to run on the ZX81, Spectrum, BBC, TRS-80, Apple, VIC 20 and PET. *"Highly recommended to anyone of any age."* Computing Today

Practical Things to do with a Microcomputer Lots of programs to run and a robot to build which will work with most micros.

Computer Jargon An illustrated guide to all the jargon.

Computer Graphics Superbly illustrated introduction to computer graphics with programs and a graphics conversion chart for most micros.

Write Your Own Adventure Programs Step-by-step guide to writing adventure games programs, with lots of expert's tips.

Machine Code for Beginners A really simple introduction to machine code for the Z80 and 6502.

Better BASIC A beginner's guide to writing programs in BASIC.

Inside the Chip A simple and colourful account of how the chip works and what it can do.

+001.99

ISBN 0-86020-735-8

00199

9 780860 207351

ISBN 0 86020 735 8

£1.99